Previous: Overview of SAAJ **Next**: Code Examples

SAAJ Tutorial

This tutorial walks you through how to use the SAAJ API. First, it covers the basics of creating and sending a simple SOAP message. Then you will learn more details about adding content to messages, including how to create SOAP faults and attributes. Finally, you will learn how to send a message and retrieve the content of the response.

After going through this tutorial, you will know how to perform the following tasks:

- Creating and Sending a Simple Message
- Adding Content to the Header
- Adding Content to the `SOAPPart` Object
- Adding a Document to the SOAP Body
- Manipulating Message Content Using SAAJ or DOM APIs
- Adding Attachments
- Adding Attributes
- Using SOAP Faults

In the section Code Examples, you will see the code fragments from earlier parts of the tutorial in runnable applications, which you can test yourself. To see how the SAAJ API can be used in server code, see the SAAJ part of the Coffee Break case study (SAAJ Coffee Supplier Service), which shows an example of both the client and the server code for a web service application.

A SAAJ client can send request-response messages to web services that are implemented to do request-response messaging. This section demonstrates how you can do this.

Creating and Sending a Simple Message

This section covers the basics of creating and sending a simple message and retrieving the content of the response. It includes the following topics:

- Creating a Message
- Parts of a Message
- Accessing Elements of a Message
- Adding Content to the Body
- Getting a `SOAPConnection` Object
- Sending a Message
- Getting the Content of a Message

Creating a Message

The first step is to create a message using a `MessageFactory` object. The SAAJ API provides a default implementation of the `MessageFactory` class, thus making it easy to get an instance. The following code fragment illustrates getting an instance of the default message factory and then using it to create a message.

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

As is true of the `newInstance` method for `SOAPConnectionFactory` , the `newInstance` method for `MessageFactory` is static, so you invoke it by calling `MessageFactory.newInstance` .

If you specify no arguments to the `newInstance` method, it creates a message factory for SOAP 1.1 messages. To create a message factory that allows you to create and process SOAP 1.2 messages, use the following method call:

```
MessageFactory factory =
    MessageFactory.newInstance(SOAPConstants.SOAP_1_2_PROTOCOL);
```

To create a message factory that can create either SOAP 1.1 or SOAP 1.2 messages, use the following method call:

```
MessageFactory factory =
    MessageFactory.newInstance(SOAPConstants.DYNAMIC_SOAP_PROTOCOL);
```

This kind of factory enables you to process an incoming message that might be of either type.

Parts of a Message

A `SOAPMessage` object is required to have certain elements, and, as stated previously, the SAAJ API simplifies things for you by returning a new `SOAPMessage` object that already contains these elements. When you call `createMessage` with no arguments, the message that is created automatically has the following:

```
I. A  SOAPPart  object that contains
    A.  A  SOAPEnvelope  object that contains
        1.  An empty  SOAPHeader  object
        2.  An empty  SOAPBody  object
```

The `SOAPHeader` object is optional and can be deleted if it is not needed. However, if there is one, it must precede the `SOAPBody` object. The `SOAPBody` object can hold either the content of the message or a **fault** message that contains status information or details about a problem with the message. The section Using SOAP Faults walks you through how to use `SOAPFault` objects.

Accessing Elements of a Message

The next step in creating a message is to access its parts so that content can be added. There are two ways to do this. The `SOAPMessage` object `message` , created in the preceding code fragment, is the place to start.

The first way to access the parts of the message is to work your way through the structure of the message. The message contains a `SOAPPart` object, so you use the `getSOAPPart` method of `message` to retrieve it:

```
SOAPPart soapPart = message.getSOAPPart();
```

Next you can use the `getEnvelope` method of `soapPart` to retrieve the `SOAPEnvelope` object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use the `getHeader` and `getBody` methods of `envelope` to retrieve its empty `SOAPHeader` and `SOAPBody` objects.

```
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

The second way to access the parts of the message is to retrieve the message header and body directly, without retrieving the `SOAPPart` or `SOAPEnvelope` . To do so, use the `getSOAPHeader` and `getSOAPBody` methods of `SOAPMessage` :

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

This example of a SAAJ client does not use a SOAP header, so you can delete it. (You will see more about headers later.) Because all `SOAPElement` objects, including `SOAPHeader` objects, are derived from the `Node` interface, you use the method `Node.detachNode` to delete `header` .

```
header.detachNode();
```

Adding Content to the Body

The `SOAPBody` object contains either content or a fault. To add content to the body, you normally create one or more `SOAPBodyElement` objects to hold the content. You can also add subelements to the `SOAPBodyElement` objects by using the `addChildElement` method. For each element or child element, you add content by using the `addTextNode` method.

When you create any new element, you also need to create an associated `javax.xml.namespace.QName` object so that it is uniquely identified.

Note –

You can use `Name` objects instead of `QName` objects. `Name` objects are specific to the SAAJ API, and you create them using either `SOAPEnvelope` methods or `SOAPFactory` methods. However, the `Name` interface may be deprecated at a future release.

The `SOAPFactory` class also lets you create XML elements when you are not creating an entire message or do not have access to a complete `SOAPMessage` object. For example, JAX-RPC implementations often work with XML fragments rather than complete `SOAPMessage` objects. Consequently, they do not have access to a `SOAPEnvelope` object, and this makes using a `SOAPFactory` object to create `Name` objects very useful. In addition to a method for creating `Name` objects, the `SOAPFactory` class provides methods for creating `Detail` objects and

SOAP fragments. You will find an explanation of `Detail` objects in Overview of SOAP Faults and Creating and Populating a `SOAPFault` Object.

`QName` objects associated with `SOAPBodyElement` or `SOAPHeaderElement` objects must be fully qualified; that is, they must be created with a namespace URI, a local part, and a namespace prefix. Specifying a namespace for an element makes clear which one is meant if more than one element has the same local name.

The following code fragment retrieves the `SOAPBody` object `body` from `message` , constructs a `QName` object for the element to be added, and adds a new `SOAPBodyElement` object to `body` .

```
SOAPBody body = message.getSOAPBody();
QName bodyName = new QName("http://wombat.ztrade.com", "GetLastTradePrice", "m");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

At this point, `body` contains a `SOAPBodyElement` object identified by the `QName` object `bodyName` , but there is still no content in `bodyElement` . Assuming that you want to get a quote for the stock of Sun Microsystems, Inc., you need to create a child element for the symbol using the `addChildElement` method. Then you need to give it the stock symbol using the `addTextNode` method. The `QName` object for the new `SOAPElement` object `symbol` is initialized with only a local name because child elements inherit the prefix and URI from the parent element.

```
QName name = new QName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

You might recall that the headers and content in a `SOAPPart` object must be in XML format. The SAAJ API takes care of this for you, building the appropriate XML constructs automatically when you call methods such as `addBodyElement` , `addChildElement` , and `addTextNode` . Note that you can call the method `addTextNode` only on an element such as `bodyElement` or any child elements that are added to it. You cannot call `addTextNode` on a `SOAPHeader` or `SOAPBody` object because they contain elements and not text.

The content that you have just added to your `SOAPBody` object will look like the following when it is sent over the wire:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="http://wombat.ztrade.com">
      <symbol>SUNW</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let’s examine this XML excerpt line by line to see how it relates to your SAAJ code. Note that an XML parser does not care about indentations, but they are generally used to indicate element levels and thereby make it easier for a human reader to understand.

Here is the SAAJ code:

```
SOAPMessage message = messageFactory.createMessage();
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

Here is the XML it produces:

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

The outermost element in this XML example is the SOAP envelope element, indicated by `SOAP-ENV:Envelope` . Note that `Envelope` is the name of the element, and `SOAP-ENV` is the namespace prefix. The interface `SOAPEnvelope` represents a SOAP envelope.

The first line signals the beginning of the SOAP envelope element, and the last line signals the end of it; everything in between is part of the SOAP envelope. The second line is an example of an attribute for the SOAP envelope element. Because a SOAP envelope element always contains this attribute with this value, a `SOAPMessage` object comes with it automatically included. `xmlns` stands for “XML namespace,” and its value is the URI of the namespace associated with `Envelope` .

The next line is an empty SOAP header. You could remove it by calling `header.detachNode` after the `getSOAPHeader` call.

The next two lines mark the beginning and end of the SOAP body, represented in SAAJ by a `SOAPBody` object. The next step is to add content to the body.

Here is the SAAJ code:

```
QName bodyName = new QName("http://wombat.ztrade.com",
     "GetLastTradePrice", "m");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

Here is the XML it produces:

```
<m:GetLastTradePrice
 xmlns:m="http://wombat.ztrade.com">
  ...
```

```
</m:GetLastTradePrice>
```

These lines are what the `SOAPBodyElement bodyElement` in your code represents. `GetLastTradePrice` is its local name, `m` is its namespace prefix, and `http://wombat.ztrade.com` is its namespace URI.

Here is the SAAJ code:

```
QName name = new QName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

Here is the XML it produces:

```
<symbol>SUNW</symbol>
```

The `String "SUNW"` is the text node for the element `<symbol>`. This `String` object is the message content that your recipient, the stock quote service, receives.

The following example shows how to add multiple `SOAPElement` objects and add text to each of them. The code first creates the `SOAPBodyElement` object `purchaseLineItems`, which has a fully qualified name associated with it. That is, the `QName` object for it has a namespace URI, a local name, and a namespace prefix. 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, can have `Name` objects with only the local name.

```
SOAPBody body = message.getSOAPBody();
QName bodyName =
    new QName("http://sonata.fruitsgalore.com", "PurchaseLineItems", "PO");
SOAPBodyElement purchaseLineItems =
    body.addBodyElement(bodyName);

QName childName = new QName("Order");
SOAPElement order = purchaseLineItems.addChildElement(childName);

childName = new QName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = new QName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = new QName("Order");
SOAPElement order2 = purchaseLineItems.addChildElement(childName);

childName = new QName("Product");
SOAPElement product2 = order2.addChildElement(childName);
product2.addTextNode("Peach");

childName = soapFactory.new QName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");
```

The SAAJ code in the preceding example produces the following XML in the SOAP body:

```
<PO:PurchaseLineItems
 xmlns:PO="http://sonata.fruitsgalore.com">
  <Order>
    <Product>Apple</Product>
    <Price>1.56</Price>
  </Order>

  <Order>
    <Product>Peach</Product>
    <Price>1.48</Price>
  </Order>
</PO:PurchaseLineItems>
```

Getting a `SOAPConnection` Object

The SAAJ API is focused primarily on reading and writing messages. After you have written a message, you can send it using various mechanisms (such as JMS or JAXM). The SAAJ API does, however, provide a simple mechanism for request-response messaging.

To send a message, a SAAJ client can use a `SOAPConnection` object. A `SOAPConnection` object is a point-to-point connection, meaning that it goes directly from the sender to the destination (usually a URL) that the sender specifies.

The first step is to obtain a `SOAPConnectionFactory` object that you can use to create your connection. The SAAJ API makes this easy by providing the `SOAPConnectionFactory` class with a default implementation. You can get an instance of this implementation using the following line of code.

```
SOAPConnectionFactory soapConnectionFactory =
    SOAPConnectionFactory.newInstance();
```

Now you can use `soapConnectionFactory` to create a `SOAPConnection` object.

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

You will use `connection` to send the message that you created.

Sending a Message

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` .

```
java.net.URL endpoint = new URL("http://wombat.ztrade.com/quotes");

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

The content of the message you sent is the stock symbol SUNW; the `SOAPMessage` object `response` should contain the last stock price for Sun Microsystems, which you will retrieve in the next section.

A connection uses a fair amount of resources, so it is a good idea to close a connection as soon as you are finished using it.

```
connection.close();
```

Getting the Content of a Message

The initial steps for retrieving a message's content are the same as those for giving content to a message: Either you use the `Message` object to get the `SOAPBody` object, or you access the `SOAPBody` object through the `SOAPPart` and `SOAPEnvelope` objects.

Then you access the `SOAPBody` object's `SOAPBodyElement` object, because that is the element to which content was added in the example. (In a later section you will see how to add content directly to the `SOAPPart` object, in which case you would not need to access the `SOAPBodyElement` object to add content or to retrieve it.)

To get the content, which was added with the method `SOAPElement.addTextNode` , you call the method `Node.getValue` . Note that `getValue` returns the value of the immediate child of the element that calls the method. Therefore, in the following code fragment, the `getValue` method is called on `bodyElement` , the element on which the `addTextNode` method was called.

To access `bodyElement` , you call the `getChildElements` method on `soapBody` . Passing `bodyName` to `getChildElements` returns a `java.util.Iterator` object that contains all the child elements identified by the `Name` object `bodyName` . You already know that there is only one, so calling the `next` method on it will return the `SOAPBodyElement` you want. Note that the `Iterator.next` method returns a Java `Object` , so you need to cast the `Object` it returns to a `SOAPBodyElement` object before assigning it to the variable `bodyElement` .

```
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);
```

If more than one element had the name `bodyName` , you would have to use a `while` loop using the `Iterator.hasNext` method to make sure that you got all of them.

```
while (iterator.hasNext()) {
    SOAPBodyElement bodyElement = (SOAPBodyElement)iterator.next();
    String lastPrice = bodyElement.getValue();
    System.out.print("The last price for SUNW is ");
    System.out.println(lastPrice);
}
```

At this point, you have seen how to send a very basic request-response message and get the content from the response. The next sections provide more detail on adding content to messages.

Adding Content to the Header

To add content to the header, you create a `SOAPHeaderElement` object. As with all new elements, it must have an associated `QName` 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();
QName headerName = new QName("http://ws-i.org/schemas/conformanceClaim/",
    "Claim", "wsi");
SOAPHeaderElement headerElement =
    header.addHeaderElement(headerName);
headerElement.addAttribute(new QName("conformsTo"),
    "http://ws-i.org/profiles/basic/1.1/");
```

At this point, `header` contains the `SOAPHeaderElement` object `headerElement` identified by the `QName` object `headerName`. Note that the `addHeaderElement` method both creates `headerElement` and adds it to `header`.

A conformance claim header has no content. This code produces the following XML header:

```
<SOAP-ENV:Header>
  <wsi:Claim
    xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
    conformsTo="http://ws-i.org/profiles/basic/1.1/"/>
</SOAP-ENV:Header>
```

For more information about creating SOAP messages that conform to WS-I, see the Conformance Claim Attachment Mechanisms document described in the Conformance section of the WS-I Basic Profile.

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 the `SOAPPart` Object

If the content you want to send is in a file, SAAJ provides an easy way to add it directly to the `SOAPPart` object. This means that you do not access the `SOAPBody` object and build the XML content yourself, as you did in the preceding section.

To add a file directly to the `SOAPPart` object, you use a `javax.xml.transform.Source` object from JAXP (the Java API for XML Processing). There are three types of `Source` objects: `SAXSource`, `DOMSource`, and `StreamSource`. A `StreamSource` object holds an XML document in text form. `SAXSource` and `DOMSource` objects hold content along with the instructions for transforming the content into an XML document.

The following code fragment uses the JAXP API to build a `DOMSource` object that is passed to the `SOAPPart.setContent` method. The first three lines of code get a `DocumentBuilderFactory` object and use it to create the `DocumentBuilder` object `builder`. Because SOAP messages use namespaces, you should set the `NamespaceAware` property for the factory to true. Then `builder` parses the content file to produce a `Document` object.

```
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document = builder.parse("file:///music/order/soap.xml");
DOMSource domSource = new DOMSource(document);
```

The following two lines of code access the `SOAPPart` object (using the `SOAPMessage` object `message`) and set the new `Document` object as its content. The `SOAPPart.setContent` method not only sets content for the `SOAPBody` object but also sets the appropriate header for the `SOAPHeader` object.

```
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

The XML file you use to set the content of the `SOAPPart` object must include `Envelope` and `Body` elements:

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

You will see other ways to add content to a message in the sections Adding a Document to the SOAP Body and Adding Attachments.

Adding a Document to the SOAP Body

In addition to setting the content of the entire SOAP message to that of a `DOMSource` object, you can add a DOM document directly to the body of the message. This capability means that you do not have to create a `javax.xml.transform.Source` object. After you parse the document, you can add it directly to the message body:

```
SOAPBody body = message.getSOAPBody();
SOAPBodyElement docElement = body.addDocument(document);
```

Manipulating Message Content Using SAAJ or DOM APIs

Because SAAJ nodes and elements implement the DOM `Node` and `Element` interfaces, you have many options for adding or changing message content:

- Use only DOM APIs.
- Use only SAAJ APIs.
- Use SAAJ APIs and then switch to using DOM APIs.

- Use DOM APIs and then switch to using SAAJ APIs.

The first three of these cause no problems. After you have created a message, whether or not you have imported its content from another document, you can start adding or changing nodes using either SAAJ or DOM APIs.

But if you use DOM APIs and then switch to using SAAJ APIs to manipulate the document, any references to objects within the tree that were obtained using DOM APIs are no longer valid. If you must use SAAJ APIs after using DOM APIs, you should set all your DOM typed references to null, because they can become invalid. For more information about the exact cases in which references become invalid, see the SAAJ API documentation.

The basic rule is that you can continue manipulating the message content using SAAJ APIs as long as you want to, but after you start manipulating it using DOM, you should no longer use SAAJ APIs.

Adding Attachments

An `AttachmentPart` object can contain any type of content, including XML. And because the SOAP part can contain only XML content, you must use an `AttachmentPart` object for any content that is not in XML format.

Creating an `AttachmentPart` Object and Adding Content

The `SOAPMessage` object creates an `AttachmentPart` object, and the message also must add the attachment to itself after content has been added. The `SOAPMessage` class has three methods for creating an `AttachmentPart` object.

The first method creates an attachment with no content. In this case, an `AttachmentPart` method is used later to add content to the attachment.

```
AttachmentPart attachment = message.createAttachmentPart();
```

You add content to `attachment` by using the `AttachmentPart` method `setContent` . This method takes two parameters: a Java `Object` for the content, and a `String` object for the MIME content type that is used to encode the object. Content in the `SOAPBody` part of a message automatically has a `Content-Type` header with the value `"text/xml"` because the content must be in XML. In contrast, the type of content in an `AttachmentPart` object must be specified because it can be any type.

Each `AttachmentPart` object has one or more MIME headers associated with it. When you specify a type to the `setContent` method, that type is used for the header `Content-Type` . Note that `Content-Type` is the only header that is required. You may set other optional headers, such as `Content-Id` and `Content-Location` . For convenience, SAAJ provides `get` and `set` methods for the headers `Content-Type` , `Content-Id` , and `Content-Location` . These headers can be helpful in accessing a particular attachment when a message has multiple attachments. For example, to access the attachments that have particular headers, you can call the `SOAPMessage` method `getAttachments` and pass it a `MIMEHeaders` object containing the MIME headers you are interested in.

The following code fragment shows one of the ways to use the method `setContent` . The Java `Object` in the first parameter can 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 must add it to the `SOAPMessage` object, something that is done in the last line.

```
String stringContent = "Update address for Sunny Skies " +
    "Inc., to 10 Upbeat Street, Pleasant Grove, CA 95439";

attachment.setContent(stringContent, "text/plain");
attachment.setContentId("update_address");

message.addAttachmentPart(attachment);
```

The `attachment` variable now represents an `AttachmentPart` object that contains the string `stringContent` and has a header that contains the string `text/plain` . It also has a `Content-Id` header with `update_address` as its value. And `attachment` is now part of `message` .

The other two `SOAPMessage.createAttachment` methods create an `AttachmentPart` object complete with content. One is very similar to the `AttachmentPart.setContent` method in that it takes the same parameters and does essentially the same thing. It takes a Java `Object` containing the content and a `String` giving the content type. As with `AttachmentPart.setContent` , the `Object` can be a `String` , a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object.

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:

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment = message.createAttachmentPart(dataHandler);
attachment.setContentId("attached_image");

message.addAttachmentPart(attachment);
```

You might note two things about this code fragment. First, it sets a header for `Content-ID` using 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, something that is possible because one of the things a `DataHandler` object does is to determine the data type of the file it contains.

Accessing an `AttachmentPart` Object

If you receive a message with attachments or want to change an attachment to a message you are building, you need to access the attachment. The `SOAPMessage` class provides two versions of the `getAttachments` method for retrieving its `AttachmentPart` objects. When it is given no argument, the method `SOAPMessage.getAttachments` returns a `java.util.Iterator` object over all the `AttachmentPart` objects in a message. When `getAttachments` is given a `MimeHeaders` object, which is a list of MIME headers, `getAttachments` returns an iterator over the `AttachmentPart` objects that have a header that matches one of the headers in the list. The following code uses the `getAttachments` method that takes no arguments and thus retrieves all the `AttachmentPart` objects in the `SOAPMessage` object `message`. Then it prints the content ID, the content type, and the content of each `AttachmentPart` object.

```
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.equals("text/plain")) {
        Object content = attachment.getContent();
        System.out.println("Attachment contains:\n" + content);
    }
}
```

Adding Attributes

An XML element can have one or more attributes that give information about that element. An attribute consists of a name for the attribute followed immediately by an equal sign (`=`) and its value.

The `SOAPElement` interface provides methods for adding an attribute, for getting the value of an attribute, and for removing an attribute. For example, in the following code fragment, the attribute named `id` is added to the `SOAPElement` object `person`. Because `person` is a `SOAPElement` object rather than a `SOAPBodyElement` object or `SOAPHeaderElement` object, it is legal for its `QName` object to contain only a local name.

```
QName attributeName = new QName("id");
person.addAttribute(attributeName, "Person7");
```

These lines of code will generate the first line in the following XML fragment.

```
<person id="Person7">
  ...
</person>
```

The following line of code retrieves the value of the attribute whose name is `id`.

```
String attributeValue = person.getAttributeValue(attributeName);
```

If you had added two or more attributes to `person`, the preceding line of code would have returned only the value for the attribute named `id`. If you wanted to retrieve the values for all the attributes for `person`, you would use the method `getAllAttributes`, which returns an iterator over all the values. The following lines of code retrieve and print each value on a separate line until there are no more attribute values. Note that the `Iterator.next` method returns a Java `Object`, which is cast to a `QName` object so that it can be assigned to the `QName` object `attributeName`. (The examples in DOM and DOMSource Examples use code similar to this.)

```
Iterator iterator = person.getAllAttributesAsQNames();
while (iterator.hasNext()){
    QName attributeName = (QName) iterator.next();
    System.out.println("Attribute name is " + attributeName.toString());
    System.out.println("Attribute value is " +
        element.getAttributeValue(attributeName));
}
```

The following line of code removes the attribute named `id` from `person`. The variable `successful` will be `true` if the attribute was removed successfully.

```
boolean successful = person.removeAttribute(attributeName);
```

In this section you have seen how to add, retrieve, and remove attributes. This information is general in that it applies to any element. The next section discusses attributes that can be added only to header elements.

Header Attributes

Attributes that appear in a `SOAPHeaderElement` object determine how a recipient processes a message. You can think of header attributes as offering a way to extend a message, giving information about such things as authentication, transaction management, payment, and so on. A header attribute refines the meaning of the header, whereas the header refines the meaning of the message contained in the SOAP body.

The SOAP 1.1 specification defines two attributes that can appear only in `SOAPHeaderElement` objects: `actor` and `mustUnderstand`.

The SOAP 1.2 specification defines three such attributes: `role` (a new name for `actor`), `mustUnderstand`, and `relay`.

The next sections discuss these attributes.

See Header Example for an example that uses the code shown in this section.

The `actor` Attribute

The `actor` attribute 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.

Note –

Although the SAAJ API provides the API for adding these attributes, it does not supply the API for processing them. For example, the actor attribute requires that there be an implementation such as a messaging provider service to route the message from one actor to the next.

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 can 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 four `SOAPHeaderElement` objects, each of which sets its `actor` attribute.

```
SOAPHeader header = message.getSOAPHeader();
SOAPFactory soapFactory = SOAPFactory.newInstance();

String nameSpace = "ns";
String nameSpaceURI = "http://gizmos.com/NSURI";

QName order = new QName(nameSpaceURI, "orderDesk", nameSpace);
SOAPHeaderElement orderHeader = header.addHeaderElement(order);
orderHeader.setActor("http://gizmos.com/orders");

QName shipping = new QName(nameSpaceURI, "shippingDesk", nameSpace);
SOAPHeaderElement shippingHeader = header.addHeaderElement(shipping);
shippingHeader.setActor("http://gizmos.com/shipping");

QName confirmation = new QName(nameSpaceURI, "confirmationDesk", nameSpace);
SOAPHeaderElement confirmationHeader = header.addHeaderElement(confirmation);
confirmationHeader.setActor("http://gizmos.com/confirmations");

QName billing = new QName(nameSpaceURI, "billingDesk", nameSpace);
SOAPHeaderElement billingHeader = header.addHeaderElement(billing);
billingHeader.setActor("http://gizmos.com/billing");
```

The `SOAPHeader` interface provides two methods that return a `java.util.Iterator` object over all the `SOAPHeaderElement` objects that have an actor that matches the specified actor. The first method, `examineHeaderElements`, returns an iterator over all the elements that have the specified actor.

```
java.util.Iterator headerElements =
    header.examineHeaderElements("http://gizmos.com/orders");
```

The second method, `extractHeaderElements`, not only returns an iterator over all the `SOAPHeaderElement` objects that have the specified actor attribute but also detaches them from the `SOAPHeader` object. So, for example, after the order desk application did its work, it would call `extractHeaderElements` to remove all the `SOAPHeaderElement` objects that applied to it.

```
java.util.Iterator headerElements =
    header.extractHeaderElements("http://gizmos.com/orders");
```

Each `SOAPHeaderElement` object can have only one actor attribute, but the same actor can be an attribute for multiple `SOAPHeaderElement` objects.

Two additional `SOAPHeader` methods, `examineAllHeaderElements` and `extractAllHeaderElements`, allow you to examine or extract all the header elements, whether or not they have an actor attribute. For example, you could use the following code to display the values of all the header elements:

```
Iterator allHeaders = header.examineAllHeaderElements();
while (allHeaders.hasNext()) {
    SOAPHeaderElement headerElement = (SOAPHeaderElement)allHeaders.next();
    QName headerName = headerElement.getElementQName();
    System.out.println("\nHeader name is " + headerName.toString());
    System.out.println("Actor is " + headerElement.getActor());
}
```

The `role` Attribute

The `role` attribute is the name used by the SOAP 1.2 specification for the SOAP 1.2 `actor` attribute. The `SOAPHeaderElement` methods `setRole` and `getRole` perform the same functions as the `setActor` and `getActor` methods.

The `mustUnderstand` Attribute

The other attribute that must be added only to a `SOAPHeaderElement` object is `mustUnderstand`. This attribute says whether or not the recipient (indicated by the `actor` attribute) is required to process a header entry. When the value of the `mustUnderstand` attribute is `true`, the actor must understand the semantics of the header entry and must process it correctly to those semantics. If the value is `false`, processing the header entry is optional. A `SOAPHeaderElement` object with no `mustUnderstand` attribute is equivalent to one with a `mustUnderstand` attribute whose value is `false`.

The `mustUnderstand` attribute is used to call attention to the fact that the semantics in an element are different from the semantics in its parent or peer elements. This allows for robust evolution, ensuring that a change in semantics will not be silently ignored by those who may not fully understand it.

If the actor for a header that has a `mustUnderstand` attribute set to `true` cannot process the header, it must send a SOAP fault back to the sender. (See Using SOAP Faults.) The actor must not change state or cause any side effects, so that, to an outside observer, it appears that the fault was sent before any header processing was done.

For example, you could set the `mustUnderstand` attribute to `true` for the `confirmationHeader` in the code fragment in The `actor` Attribute:

```
QName confirmation = new QName(nameSpaceURI, "confirmationDesk", nameSpace);
SOAPHeaderElement confirmationHeader = header.addHeaderElement(confirmation);
confirmationHeader.setActor("http://gizmos.com/confirmations");
confirmationHeader.setMustUnderstand(true);
```

This fragment produces the following XML:

```
<ns:confirmationDesk
  xmlns:ns="http://gizmos.com/NSURI"
  SOAP-ENV:actor="http://gizmos.com/confirmations"
  SOAP-ENV:mustUnderstand="1"/>
```

You can use the `getMustUnderstand` method to retrieve the value of the `mustUnderstand` attribute. For example, you could add the following to the code fragment at the end of the preceding section:

```
System.out.println("mustUnderstand is " + headerElement.getMustUnderstand());
```

The `relay` Attribute

The SOAP 1.2 specification adds a third attribute to a `SOAPHeaderElement`, `relay`. This attribute, like `mustUnderstand`, is a boolean value. If it is set to `true`, it indicates that the SOAP header block must not be processed by any node that is targeted by the header block, but must only be passed on to the next targeted node. This attribute is ignored on header blocks whose `mustUnderstand` attribute is set to true or that are targeted at the ultimate receiver (which is the default). The default value of this attribute is `false`.

For example, you could set the `relay` element to `true` for the `billingHeader` in the code fragment in The `actor` Attribute (also changing `setActor` to `setRole`):

```
QName billing = new QName(nameSpaceURI, "billingDesk", nameSpace);
SOAPHeaderElement billingHeader = header.addHeaderElement(billing);
billingHeader.setRole("http://gizmos.com/billing");
billingHeader.setRelay(true);
```

This fragment produces the following XML:

```
<ns:billingDesk
  xmlns:ns="http://gizmos.com/NSURI"
  env:relay="true"
  env:role="http://gizmos.com/billing"/>
```

To display the value of the attribute, call `getRelay`:

```
System.out.println("relay is " + headerElement.getRelay());
```

Using SOAP Faults

In this section, you will see how to use the API for creating and accessing a SOAP fault element in an XML message.

Overview of SOAP Faults

If you send a message that was not successful for some reason, you may get back a response containing a SOAP fault element, which gives you status information, error information, or both. There can be only one SOAP fault element in a message, and it must be an entry in the SOAP body. Furthermore, if there is a SOAP fault element in the SOAP body, there can be no other elements in the SOAP body. This means that when you add a SOAP fault element, you have effectively completed the construction of the SOAP body.

A `SOAPFault` object, the representation of a SOAP fault element in the SAAJ API, is similar to an `Exception` object in that it conveys information about a problem. However, a `SOAPFault` object is quite different in that it is an element in a message's `SOAPBody` object rather than part of the `try` / `catch` mechanism used for `Exception` objects. Also, as part of the `SOAPBody` object, which provides a simple means for sending mandatory information intended for the ultimate recipient, a `SOAPFault` object only reports status or error information. It does not halt the execution of an application, as an `Exception` object can.

If you are a client using the SAAJ API and are sending point-to-point messages, the recipient of your message may add a `SOAPFault` object to the response to alert you to a problem. For example, if you sent an order with an incomplete address for where to send the order, the service receiving the order might put a `SOAPFault` object in the return message telling you that part of the address was missing.

Another example of who might send a SOAP fault is an intermediate recipient, or actor. As stated in the section Adding Attributes, an actor that cannot process a header that has a `mustUnderstand` attribute with a value of `true` must return a SOAP fault to the sender.

A `SOAPFault` object contains the following elements:

- **Fault code**: Always required. The fault code must be a fully qualified name: it must contain a prefix followed by a local name. The SOAP specifications define a set of fault code local name values, which a developer can extend to cover other problems. (These are defined in section 4.4.1 of the SOAP 1.1 specification and in section 5.4.6 of the SOAP 1.2 specification.) Table 19–1 lists and describes the default fault code local names defined in the specifications.

  A SOAP 1.2 fault code can optionally have a hierarchy of one or more subcodes.
- **Fault string**: Always required. A human-readable explanation of the fault.
- **Fault actor**: Required if the `SOAPHeader` object contains one or more `actor` attributes; optional if no actors are specified, meaning that the only actor is the ultimate destination. The fault actor, which is specified as a URI, identifies who caused the fault. For an explanation of what an actor is, see The `actor` Attribute.
- `Detail` **object**: Required if the fault is an error related to the `SOAPBody` object. If, for example, the fault code is `Client` , indicating that the message could not be processed because of a problem in the `SOAPBody` object, the `SOAPFault` object must contain a `Detail` object that gives details about the problem. If a `SOAPFault` object does not contain a `Detail` object, it can be assumed that the `SOAPBody` object was processed successfully.

Creating and Populating a `SOAPFault` Object

You have seen how to add content to a `SOAPBody` object; this section walks you through adding a `SOAPFault` object to a `SOAPBody` object and then adding its constituent parts.

As with adding content, the first step is to access the `SOAPBody` object.

```
SOAPBody body = message.getSOAPBody();
```

With the `SOAPBody` object `body` in hand, you can use it to create a `SOAPFault` object. The following line of code creates a `SOAPFault` object and adds it to `body` .

```
SOAPFault fault = body.addFault();
```

The `SOAPFault` interface provides convenience methods that create an element, add the new element to the `SOAPFault` object, and add a text node, all in one operation. For example, in the following lines of SOAP 1.1 code, the method `setFaultCode` creates a `faultcode` element, adds it to `fault` , and adds a `Text` node with the value `"SOAP-ENV:Server"` by specifying a default prefix and the namespace URI for a SOAP envelope.

```
QName faultName = new QName(SOAPConstants.URI_NS_SOAP_ENVELOPE, "Server");
fault.setFaultCode(faultName);
fault.setFaultActor("http://gizmos.com/orders");
fault.setFaultString("Server not responding");
```

The SOAP 1.2 code would look like this:

```
QName faultName = new QName(SOAPConstants.URI_NS_SOAP_1_2_ENVELOPE, "Receiver");
fault.setFaultCode(faultName);
fault.setFaultRole("http://gizmos.com/order");
fault.addFaultReasonText("Server not responding", Locale.US);
```

To add one or more subcodes to the fault code, call the method `fault.appendFaultSubcode` , which takes a `QName` argument.

The `SOAPFault` object `fault` , created in the preceding lines of code, indicates that the cause of the problem is an unavailable server and that the actor at `http://gizmos.com/orders` is having the problem. If the message were being routed only to its ultimate destination, there would have been no need to set a fault actor. Also note that `fault` does not have a `Detail` object because it does not relate to the `SOAPBody` object. (If you use SOAP 1.2, you can use the `setFaultRole` method instead of `setFaultActor` .)

The following SOAP 1.1 code fragment creates a `SOAPFault` object that includes a `Detail` object. Note that a `SOAPFault` object can have only one `Detail` object, which is simply a container for `DetailEntry` objects, but the `Detail` object can have multiple `DetailEntry` objects. The `Detail` object in the following lines of code has two `DetailEntry` objects added to it.

```
SOAPFault fault = body.addFault();

QName faultName = new QName(SOAPConstants.URI_NS_SOAP_ENVELOPE, "Client");
fault.setFaultCode(faultName);
fault.setFaultString("Message does not have necessary info");

Detail detail = fault.addDetail();
```

```
QName entryName = new QName("http://gizmos.com/orders/", "order", "PO");
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("Quantity element does not have a value");

QName entryName2 = new QName("http://gizmos.com/orders/", "order", "PO");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

See SOAP Fault Example for an example that uses code like that shown in this section.

The SOAP 1.1 and 1.2 specifications define slightly different values for a fault code. Table 19–1 lists and describes these values.

Table 19–1 SOAP Fault Code Values

SOAP 1.1	SOAP 1.2	Description
VersionMismatch	VersionMismatch	The namespace or local name for a `SOAPEnvelope` object was invalid.
MustUnderstand	MustUnderstand	An immediate child element of a `SOAPHeader` object had its `mustUnderstand` attribute set to `true`, and the processing party did not understand the element or did not obey it.
Client	Sender	The `SOAPMessage` object was not formed correctly or did not contain the information needed to succeed.
Server	Receiver	The `SOAPMessage` object could not be processed because of a processing error, not because of a problem with the message itself.
N/A	DataEncodingUnknown	A SOAP header block or SOAP body child element information item targeted at the faulting SOAP node is scoped with a data encoding that the faulting node does not support.

Retrieving Fault Information

Just as the `SOAPFault` interface provides convenience methods for adding information, it also provides convenience methods for retrieving that information. The following code fragment shows what you might write to retrieve fault information from a message you received. In the code fragment, `newMessage` is the `SOAPMessage` object that has been sent to you. Because a `SOAPFault` object must be part of the `SOAPBody` object, the first step is to access the `SOAPBody` object. Then the code tests to see whether the `SOAPBody` object contains a `SOAPFault` object. If it does, the code retrieves the `SOAPFault` object and uses it to retrieve its contents. The convenience methods `getFaultCode`, `getFaultString`, and `getFaultActor` make retrieving the values very easy.

```
SOAPBody body = newMessage.getSOAPBody();
if ( body.hasFault() ) {
    SOAPFault newFault = body.getFault();
    QName code = newFault.getFaultCodeAsQName();
    String string = newFault.getFaultString();
    String actor = newFault.getFaultActor();
```

To retrieve subcodes from a SOAP 1.2 fault, call the method `newFault.getFaultSubcodes`.

Next the code prints the values it has just retrieved. Not all messages are required to have a fault actor, so the code tests to see whether there is one. Testing whether the variable `actor` is `null` works because the method `getFaultActor` returns `null` if a fault actor has not been set.

```
    System.out.println("SOAP fault contains: ");
    System.out.println("  Fault code = " + code.toString());
    System.out.println("  Local name = " + code.getLocalPart());
    System.out.println("  Namespace prefix = " +
        code.getPrefix() + ", bound to " + code.getNamespaceURI());
    System.out.println("  Fault string = " + string);

    if ( actor != null ) {
        System.out.println("  Fault actor = " + actor);
    }
```

The final task is to retrieve the `Detail` object and get its `DetailEntry` objects. The code uses the `SOAPFault` object `newFault` to retrieve the `Detail` object `newDetail`, and then it uses `newDetail` to call the method `getDetailEntries`. This method returns the `java.util.Iterator` object `entries`, which contains all the `DetailEntry` objects in `newDetail`. Not all `SOAPFault` objects are required to have a `Detail` object, so the code tests to see whether `newDetail` is `null`. If it is not, the code prints the values of the `DetailEntry` objects as long as there are any.

```
Detail newDetail = newFault.getDetail();
if (newDetail != null) {
    Iterator entries = newDetail.getDetailEntries();
    while ( entries.hasNext() ) {
        DetailEntry newEntry = (DetailEntry)entries.next();
        String value = newEntry.getValue();
        System.out.println("  Detail entry = " + value);
    }
}
```

In summary, you have seen how to add a `SOAPFault` object and its contents to a message as well as how to retrieve the contents. A `SOAPFault` object, which is optional, is added to the `SOAPBody` object to convey status or error information. It must always have a fault code and a `String` explanation of the fault. A `SOAPFault` object must indicate the actor that is the source of the fault only when there are multiple actors; otherwise, it is optional. Similarly, the `SOAPFault` object must contain a `Detail` object with one or more `DetailEntry` objects only when the contents of the `SOAPBody` object could not be processed successfully.

See SOAP Fault Example for an example that uses code like that shown in this section.

Previous: Overview of SAAJ **Next**: Code Examples