GET INVOLVED
New to Java?
Blogs
CodeShare
Newsgroups
Wiki
User Groups
Submit Content/Code
PRODUCT CENTERS
BEA AquaLogic
AquaLogic BPM Suite
AquaLogic Service Bus
WebLogic Platform
WebLogic Server
BEA Workshop
WebLogic Portal
WebLogic Integration
BEA JRockit
WebLogic Event Server
BEA WLCP
More Product Centers
TECHNOLOGY CENTERS
Controls
Web Services
Eclipse
XML
EJB
Persistence
JMS
Vertical Markets
Dev Toolbox
BROWSE BY ROLE
Platform Admin

HOW TO USE XML MAPS

by Tom Clements, Radu Preotiuc
11/12/2002

Wouldn't it be nice to be able to handle XML data in Java code without having to immerse yourself in all the low-level details of the markup language, without having to understand XML Schemas and DTDs, without having to marshall and unmarshall data, and without ever having to deal with the complexities of SAX and DOM parsers?

If you've said "Yes" to any or all of the above, then the XML Mapping framework of WebLogic Workshop will make your job as a developer a lot easier. In Part I of this article, we talk about the different types of XML Maps, walking you through several mapping scenarios made possible by WebLogic Workshop. Mapping syntax is described along with diagrams that call out the correlations between XML content and Java method signatures. Some important mapping directives are also described. Part II of this article deals with advanced mappings that require the use of ECMAScripting.

Why Use XML Maps?
XML provides a standard way of passing data between Web Services and Web Service clients. Because the client and the Web Service only care about the data being passed, not about how the data is used, XML promotes loose coupling. The implementation can change on one side of the transfer without impacting operations on the other side.

Loose coupling of this sort, however, applies only between web services, not between a web service and the XML interface it supports. Web services are still tightly coupled to the interface. If the shape of the XML message changes, your code may break. What WebLogic Workshop provides is a mapping layer that separates out dependencies between web service code and XML messages, creating a loose-coupling between Java and XML.

With XML Maps, the contents of an XML message may be passed into Java parameters and Java return values may be passed back into XML elements and attributes. The mapping layer in WebLogic Workshop lets you specify how the XML data is converted into Java types. WebLogic Workshop then uses the XML map to generate the code necessary for the conversion.

By applying XML maps, you control the translation of XML to Java or Java to XML and preserve loosely-coupled message exchanges.

Map Types

DISCUSS
Post your questions or comments below

ARTICLE TOOLS
E-mail
Print
Blog

RELATED PRODUCTS
Check out the products mentioned in this article:
- BEA Workshop Product Family

RELATED TECHNOLOGIES
- Web Services
- XML

RELATED ARTICLES
- Using JAX-WS and JAXB with WebLogic Server 10: JAX-WS Customization Binding
- Using JAX-WS and JAXB with WebLogic Server 10
- BPEL4People

BOOKMARK ARTICLE
del.icio.us

There are three types of XML maps:

- Default
- Custom
- Scripted

Note: Scripted maps are discussed in a second article.

Default XML Maps

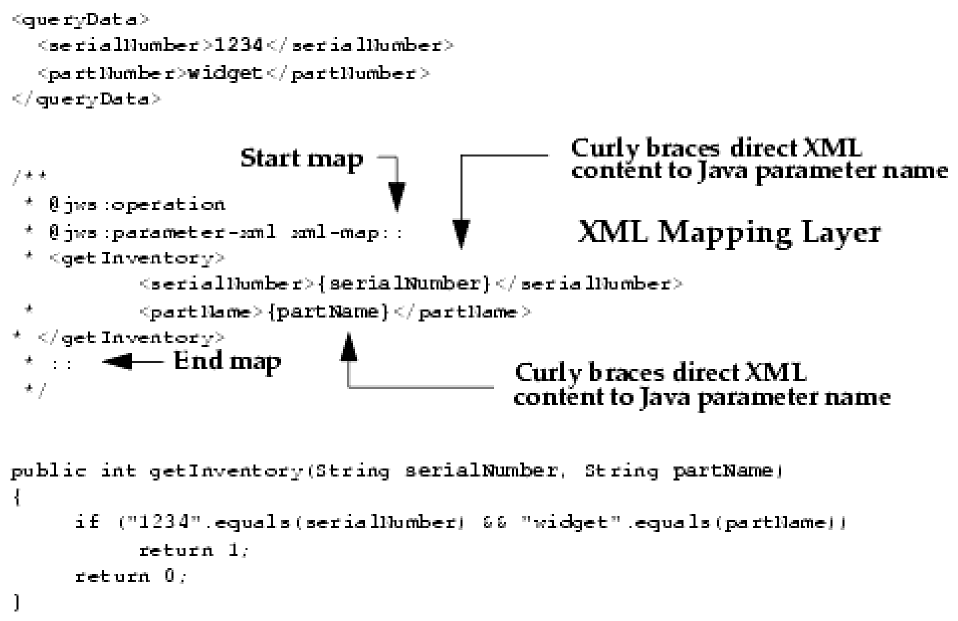
Web services communicate by sending and receiving XML messages. As long as the parameter types in the declaration of your Java web service match the elements and attributes defined in the XML message you expect, there is no need for *custom* XML mapping. Instead, WebLogic Server automatically translates these messages to and from the types in your Java declaration according to a natural, or *default*, map.

Default maps are used when there are no pre-determined constraints on the format of the XML message -- that is, when expected XML message formats conform closely to the parameters and return values of your Java method signature.

Consider the following code snippet from an XML message carrying data used to submit a request for information about a manufacturer's inventory. Notice that the serialNumber and partNumber element names of the XML message explicitly match the parameter names of the Java declaration. This is the simplest possible case. The Mapping Layer simply reiterates the Java parameter names and provides **curly braces** as placeholders for XML content.

Java parameter names are placed within curly braces to indicate *where* the XML values are to be directed. In essence, curly braces act as **substitution directives** to funnel XML content to the appropriate Java parameter.

In this example, the contents of the serialNumber element ("1234") are directed via curly braces to the Java serialNumber parameter. In the same way, the contents of the partNumber element ("widget") are directed via curly braces to the Java partNumber parameter.


Start map
Curly braces direct XML content to Java parameter name
XML Mapping Layer
End map
Curly braces direct XML content to Java parameter name

```
<queryData>
  <serialNumber>1234</serialNumber>
  <partNumber>widget</partNumber>
</queryData>

/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <getInventory>
          <serialNumber>{serialNumber}</serialNumber>
 *        <partName>{partName}</partName>
* </getInventory>
 * ::
 */

public int getInventory(String serialNumber, String partName)
{
     if ("1234".equals(serialNumber) && "widget".equals(partName))
          return 1;
     return 0;
}
```

All this typically takes place within the context of a javadoc @jws:operation. A .jws file is a Java Web Service (JWS) source code file generated by WebLogic Workshop that contains Javadoc tags that declare the properties of the web service. The method for which the map is defined (in this case, getInventory) must be marked as a JWS operation.

Note: To illustrate the details of the processing involved, the example above explicitly defines the elements of the XML mapping layer, which -- in a default map -- would normally be generated under the covers by WebLogic Server. Screen shots of actual default maps are presented later in this article.

There are two types of maps described in WebLogic Workshop:

- Parameter XML maps
- Return XML maps

Parameter XML Maps
Parameter maps are used to represent the parameters sent to the Web Service. When the client passes XML data to the Web Service, the parameter XML map defines how the data is mapped to Java types, which can then be manipulated by the Web Service.

In the code sample described above, the double colon (: :) indicates the beginning of the parameter XML map. This is followed by the <getInventory> element, which corresponds to the name of the Web Service method.

Note: Root tags for XML maps (such as <getInventory> above) must be unique across XML maps within a given JWS file.

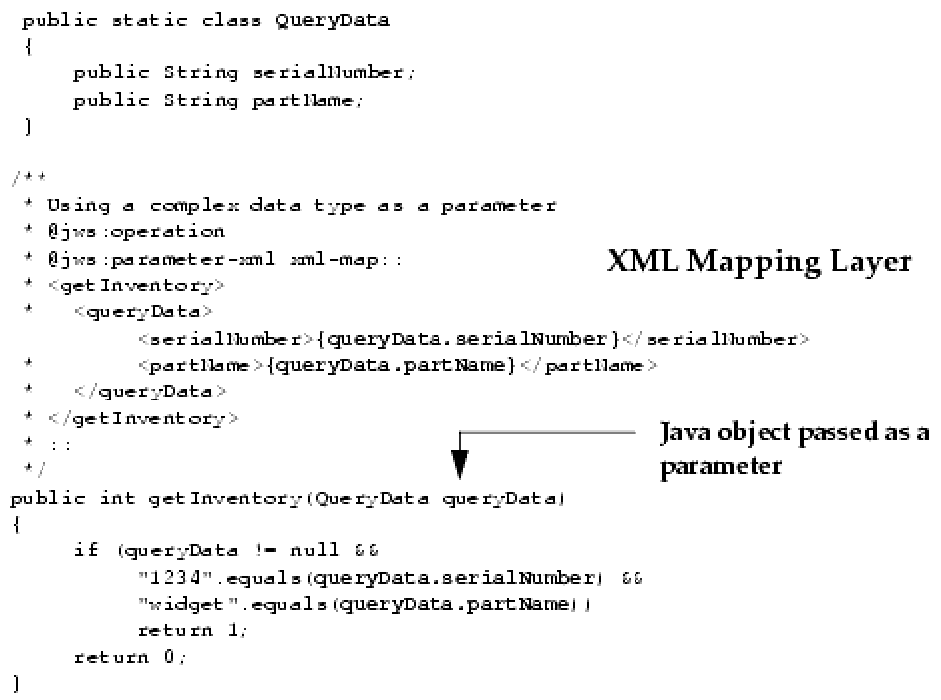
Consequently, it's recommended you place your XML map within tags whose names match your Web Service method name, since Java methods must also be unique.

The remainder of the XML map defines a queryData element and two child elements, serialNumber and partName. The contents of the child elements, as discussed previously, are represented by curly braces. It's important to note that the parameters specified between the curly braces must match the names of the parameters of the Web Service method for which the map was created.

A final double colon (: :) indicates the end of the XML map.

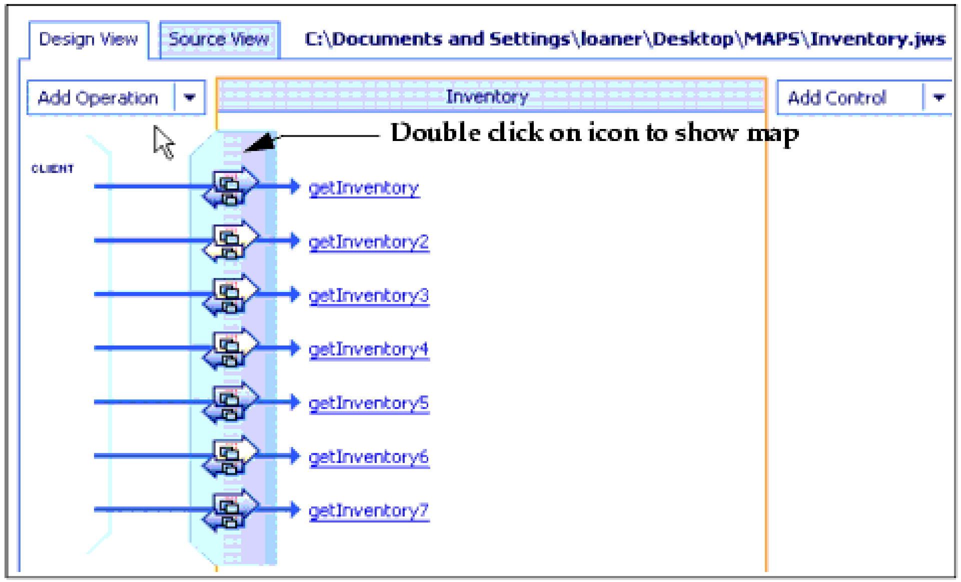
Mapping Java Objects
On occasion, you may want to pass Java objects to a Web Service method or return a Java object from a Web Service method. You can use the dot-operator to assign the values of Java data members to XML, and vice versa.

Suppose you have a Java object called QueryData with class members serialNumber and partName. You can reference the individual members in the object using dot notation, as illustrated below:


XML Mapping Layer
Java object passed as a parameter

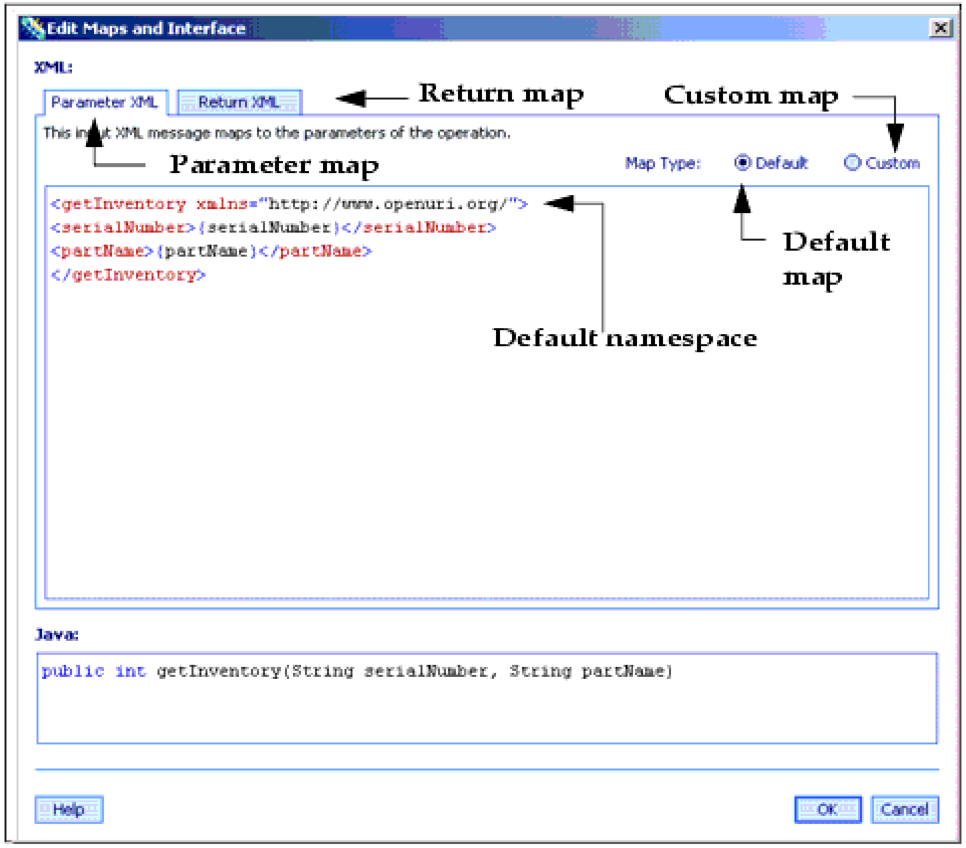
```
 public static class QueryData
 {
     public String serialNumber;
     public String partName;
 }

/**
 * Using a complex data type as a parameter
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <getInventory>
 *   <queryData>
         <serialNumber>{queryData.serialNumber}</serialNumber>
 *       <partName>{queryData.partName}</partName>
 *   </queryData>
 * </getInventory>
 * ::
 */
public int getInventory(QueryData queryData)
{
     if (queryData != null &&
          "1234".equals(queryData.serialNumber) &&
          "widget".equals(queryData.partName))
          return 1;
     return 0;
}
```

Edit Maps and Interface Dialog
The nice thing about WebLogic Workshop is that default maps are constructed automatically by the underlying framework. As a result, you don't have to build XML maps from scratch. To see an XML map from within WebLogic Workshop, simply double click on the map icon in Design View of the Workshop canvas, as illustrated below.



Note: The Inventory.jws file, which contains complete sample code for the methods displayed above, is available for download with this article.

The Edit Maps and Interface dialog box opens and displays the default XML map.



This input XML message maps to the parameters of the operation.

Map Type: [✓ Default] Custom

```
<getInventory xmlns="http://www.openuri.org/">
<serialNumber>{serialNumber}</serialNumber>
<partName>{partName}</partName>
</getInventory>
```

```
public int getInventory(String serialNumber, String partName)
```

Notice the two tabs (Parameter XML and Return XML) in the top panel of the Edit Maps and Interface dialog. The parameter map is displayed because the Parameter Map tab is selected. You can toggle between the parameter and return map views by clicking the appropriate tags.

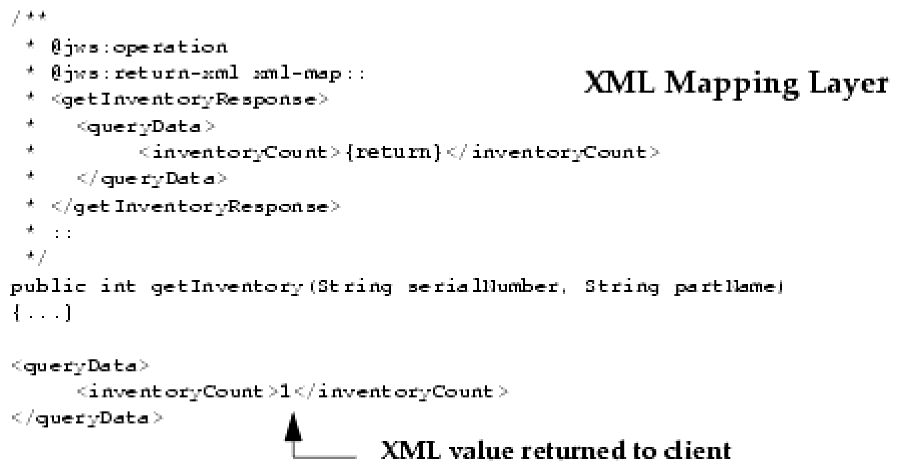
Also notice the Default and Custom radio buttons. The Default radio button displays an XML Map automatically generated by WebLogic Workshop. To create your own custom mapping layer, select the Custom button.

Finally, consider the default namespace. Namespaces provide a way for you to ensure that element names are unique within a given XML document. A namespace looks like URL but -- in fact -- it's not. It's just a way to uniquely identify XML elements. Namespaces in XML are similar to Java packages, which provide a way of differentiating between two classes with the same name. Later in this article, we'll discuss XML mapping directives such as <xm:value>, <xm:attribute>, <xm:bind> and <xm:multiple>. The *xm* prefix in these directives is a namespace qualifier.

Note: The namespace specified by the xm prefix is implicitly declared in all files where XML Maps are used but may be overridden by another prefix you define.

Return XML Maps

Similar to parameter maps, return XML maps are typically defined within the context of a javadoc @jws:operation. Data flows, however, are reversed. That is, return XML maps are used to map Java return values to XML.

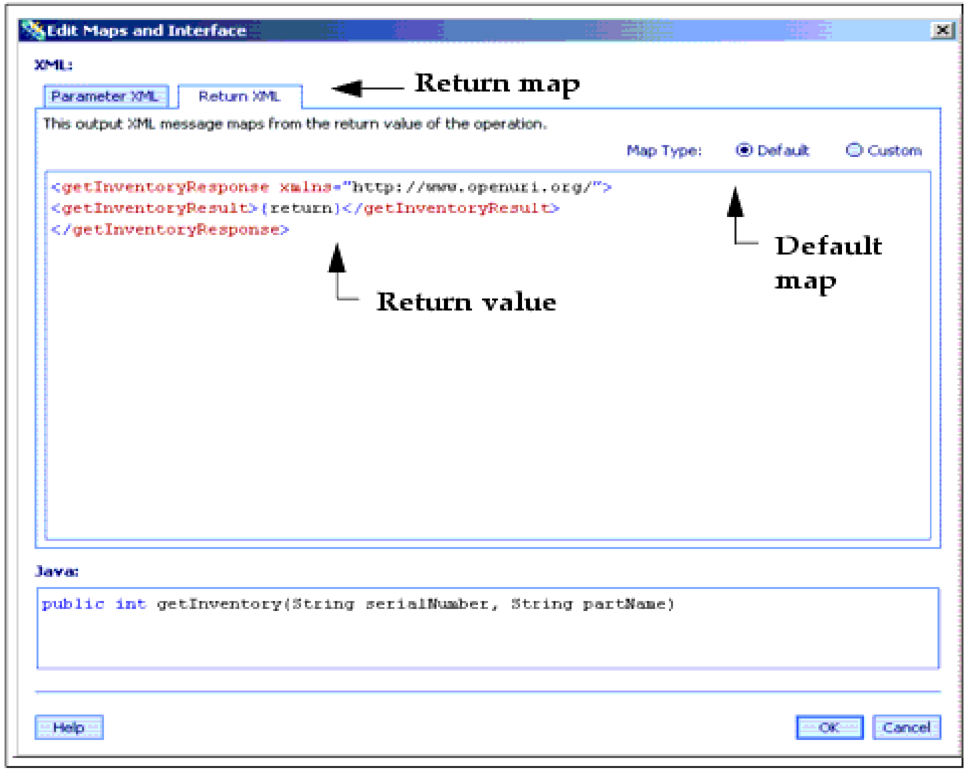


```
/**
 * @jws:operation
 * @jws:return-xml xml-map::
 * <getInventoryResponse>
 *   <queryData>
 *        <inventoryCount>{return}</inventoryCount>
 *   </queryData>
 * </getInventoryResponse>
 * ::
 */
public int getInventory(String serialNumber, String partName)
{...}

<queryData>
     <inventoryCount>1</inventoryCount>
</queryData>
```

In the example above, the method getInventory() is defined by the Web Service. This method returns an inventory count for the part requested. When the client makes a call to this method, the expected result is an integer, which is passed back to the client by WebLogic Workshop.

Recall that when mapping parameters with an parameter map, parameter names were substituted for XML values. When mapping a return value in a return map, the word **"return"** in curly braces is substituted for the XML.

For default maps, WebLogic Workshop automatically generates a return map based on the structure of the Java type, as illustrated below.



This output XML message maps from the return value of the operation.

Map Type: [✓ Default] Custom

```
<getInventoryResponse xmlns="http://www.openuri.org/">
<getInventoryResult>{return}</getInventoryResult>
</getInventoryResponse>
```

```
public int getInventory(String serialNumber, String partName)
```

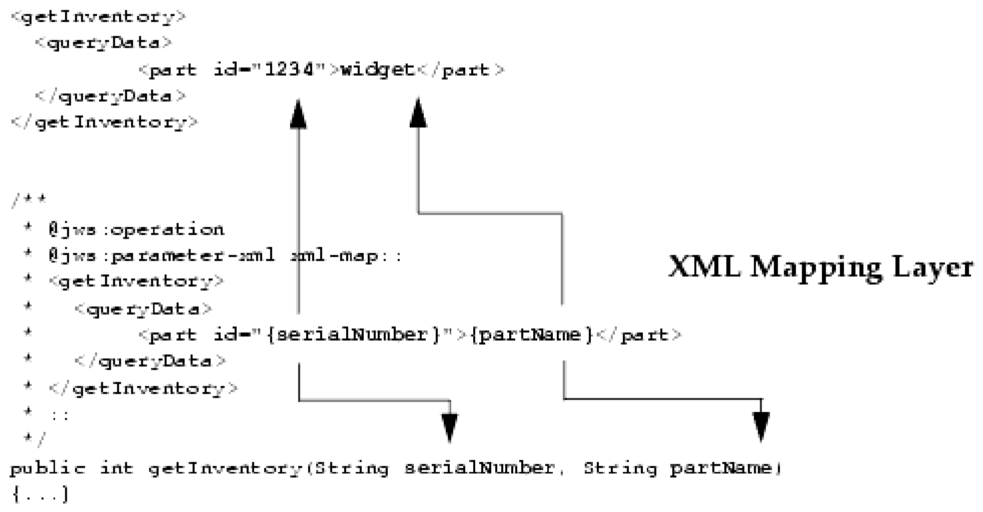
As with parameter XML maps, return XML maps can use the dot-operator in conjunction with complex data types to specify class members.

Custom XML Maps

Recall that default maps are used when the elements and attributes of an XML message conform closely to the parameters of your Java method signature. What happens when they don't? It's sometimes the case, for example, that XML messages are written to the standards of a specific industry, where element and attribute names may bear little resemblance to Java parameters. Rather than trying to retrofit your web service code to meet the industry-specific needs of the client, you can create an XML map to do the job for you.

Custom maps are therefore used when a mismatch occurs between the constraints of the expected XML message and your Java parameters and return values. Using custom maps, you eliminate dependencies between your Java code and XML.

Consider the following code snippet from an XML message carrying data used to submit a request for information about a manufacturer's inventory:



```
<getInventory>
  <queryData>
          <part id="1234">widget</part>
  </queryData>
</getInventory>


/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <getInventory>
 *   <queryData>
 *        <part id="{serialNumber}">{partName}</part>
 *   </queryData>
 * </getInventory>
 * ::
 */
public int getInventory(String serialNumber, String partName)
{...}
```

In this example, the names of the XML message attributes and elements do not correspond directly to the names of the Java parameters in the getInventory method. The XML message format, in other words, differs from the method signature of the web service designed to handle it. You do not, however, want to change your Java implementation code. Nor do you expect the client, whose XML message format may be specific to an industry, to conform to your web service implementation.

The solution, as provided by WebLogic Workshop, is to construct a mapping layer in the JWS file that lets you associate specific XML element content and XML attribute values to specific Java parameters and return values without incurring dependencies among them.

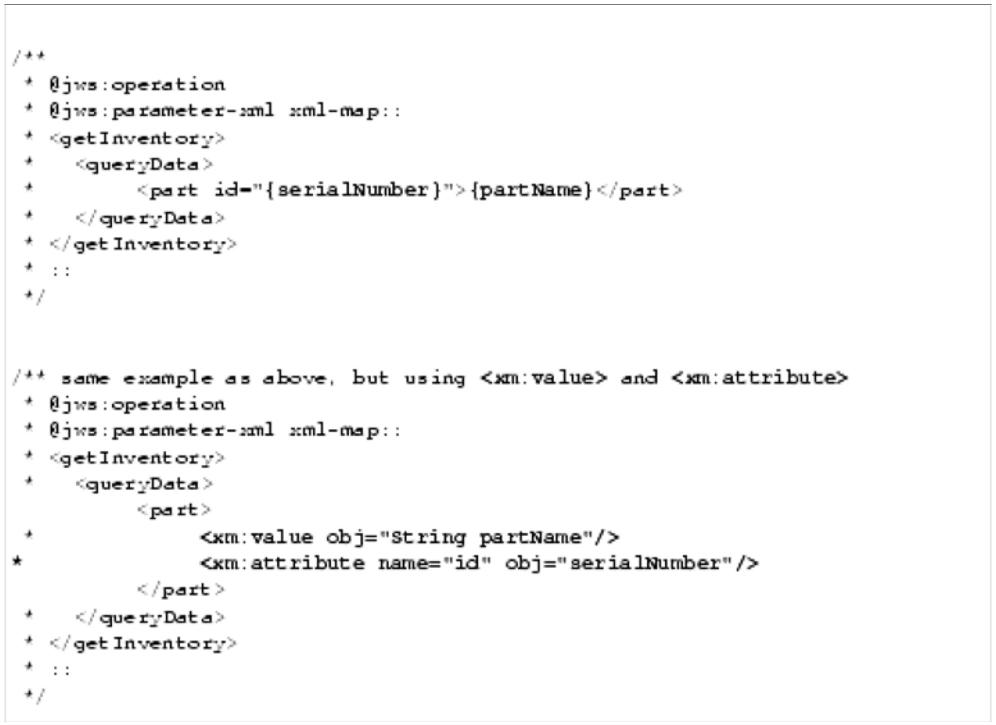
In the example above, the serialNumber parameter is mapped to the value of the id attribute, while the partName parameter is mapped to the value of the part element. This is accomplished through the use of {} (curly braces) in the Mapping Layer. As with default maps, curly braces act as *substitution directives* that funnel XML content into Java parameters.

XML Mapping Directives
In addition to the general mapping types discussed so far, there are several mapping directives that are useful in handling special mapping cases. These directives include:

- <xm: value> and <xm: attribute>
- <xm: multiple>
- <xm: bind>

<xm: value> and <xm:attribute>

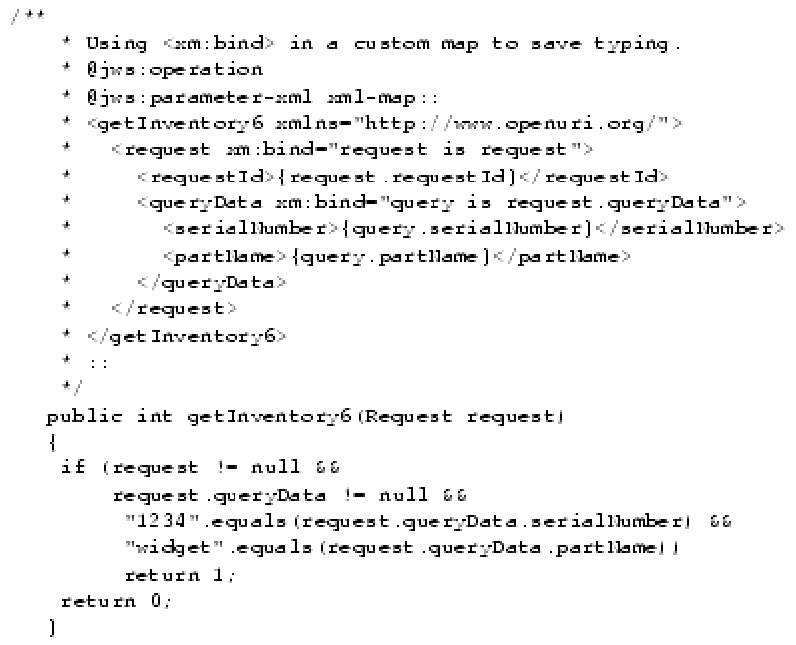
Curly braces represent a quick and convenient way (a syntactical shortcut) to map the content of an XML element or attribute to a Java parameter. The <xm:value> and <xm:attribute> directives provide a more explicit way of accomplishing the same thing. Use <xm:value> to map XML elements. Use <xm:attribute> to map XML attributes. The two examples that follow are equivalent:



```
/**
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <getInventory>
 *   <queryData>
 *        <part id="{serialNumber}">{partName}</part>
 *   </queryData>
 * </getInventory>
 * ::
 */


/** same example as above, but using <xm:value> and <xm:attribute>
 * @jws:operation
 * @jws:parameter-xml xml-map::
 * <getInventory>
 *   <queryData>
          <part>
 *             <xm:value obj="String partName"/>
*              <xm:attribute name="id" obj="serialNumber"/>
          </part>
 *   </queryData>
 * </getInventory>
 * ::
 */
```

<xm: bind>

You can declare temporary variables within an XML map using the <xm:bind> directive. This directive is helpful when you need to populate so many fields in a Java object that the value names become awkward or unwieldy. For example:



```
/**
   * Using <xm:bind> in a custom map to save typing.
   * @jws:operation
   * @jws:parameter-xml xml-map::
   * <getInventory6 xmlns="http://www.openuri.org/">
   *   <request xm:bind="request is request">
   *     <requestId>{request.requestId}</requestId>
   *     <queryData xm:bind="query is request.queryData">
   *       <serialNumber>{query.serialNumber}</serialNumber>
   *       <partName>{query.partName}</partName>
   *     </queryData>
   *   </request>
   * </getInventory6>
   * ::
   */
  public int getInventory6(Request request)
  {
   if (request != null &&
       request.queryData != null &&
        "1234".equals(request.queryData.serialNumber) &&
        "widget".equals(request.queryData.partName))
        return 1;
   return 0;
  }
```

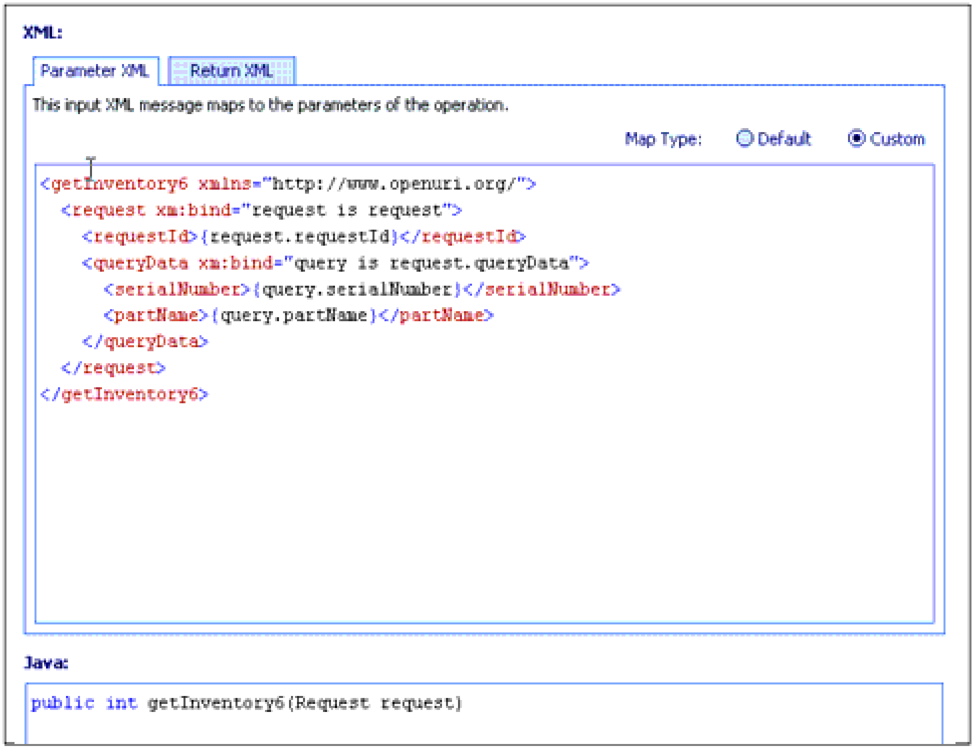
Note: An xm:bind variable is only available within the element in which it was defined.

The value of the <xm:bin> directive has the following form:

<some-element xm:bind="variable-type variable-name is value">

The variable-type is the (optional) Java class name of the object you want to bind. The variable-name is the name you want to call the variable within the XML map. The value attribute indicates what the variable actually refers to.

The custom Parameter XML map you create for this code looks like this:



This input XML message maps to the parameters of the operation.

Map Type: Default [✓ Custom]

```
<getInventory6 xmlns="http://www.openuri.org/">
  <request xm:bind="request is request">
    <requestId>{request.requestId}</requestId>
    <queryData xm:bind="query is request.queryData">
      <serialNumber>{query.serialNumber}</serialNumber>
      <partName>{query.partName}</partName>
    </queryData>
  </request>
</getInventory6>
```

```
public int getInventory6(Request request)
```

<xm: multiple>

An XML message may contain elements that occur multiple times within an XML document. For example:

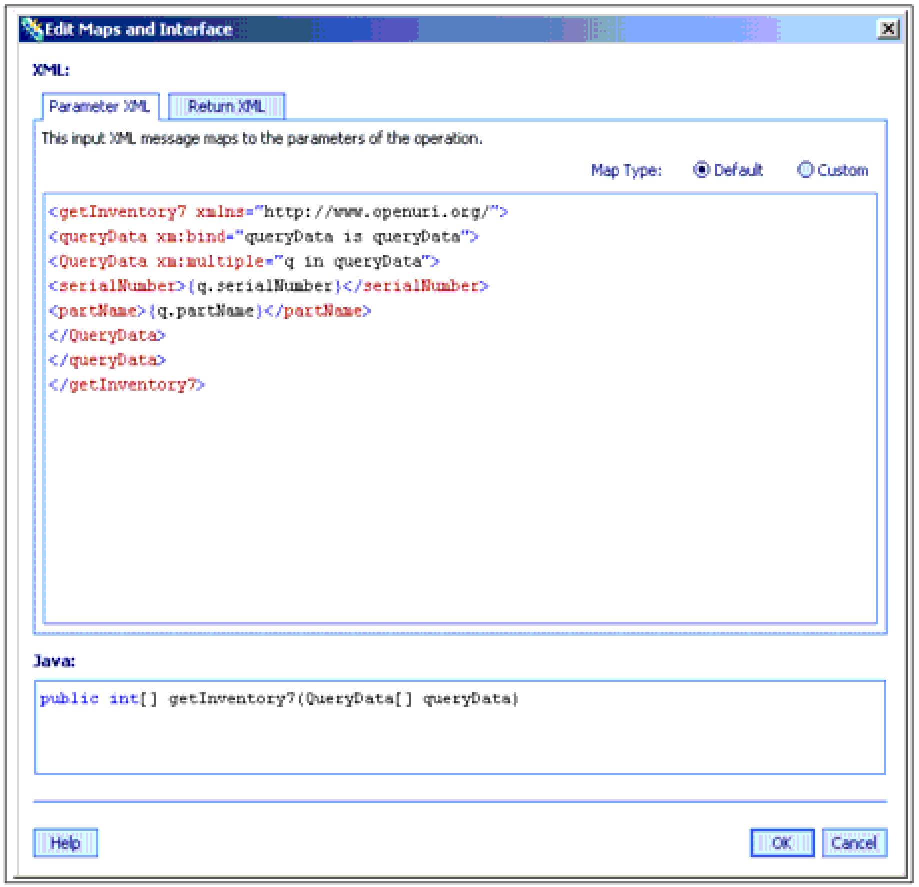
```
<QueryData>
   <serialNumber>1234</serialNumber>
   <partNumber>widget</partNumber>
</QueryData>
<QueryData>
   <serialNumber>2345</serialNumber>
   <partNumber>widget2</partNumber>
</QueryData>
```

You can handle this situation programmatically via an array in your .jws file. In the following code sample, the <QueryData> element represents an array that repeats to make up a single XML order with multiple child elements. The name of the XML element corresponds to the Java type.

```
 /**
     * Using <xm:multiple>, which is similar to <xm:bind> with the
     * exception that <xm:multiple> will bind its variable to each
     * of the values in the array, resulting in the number of blocks of the
     * included XML being equal to the number of elements in the array.
     * @jws:operation
     */
    public int[] getInventory7(QueryData[] queryData)
    {
     if (queryData == null)
          return new int[0];
     int[] result = new int[queryData.length];
     for (int i = 0; i< result.length; i++)
     {
          if (queryData[i] != null &&
               "1234".equals(queryData[i].serialNumber) &&
               "widget".equals(queryData[i].partName))
               result[i] = 1;
     }
     return result;
    }
}
```

In this example, the <xm:multiple> directive specifies that the contents of the serialNumber and partName members of the queryData array be stored as repeating blocks of XML elements.

The default Parameter map generated by WebLogic Server from the preceding code looks like this:



This input XML message maps to the parameters of the operation.

Map Type: [✓ Default] Custom

```
<getInventory7 xmlns="http://www.openuri.org/">
<queryData xm:bind="queryData is queryData">
<QueryData xm:multiple="q in queryData">
<serialNumber>{q.serialNumber}</serialNumber>
<partName>{q.partName}</partName>
</QueryData>
</queryData>
</getInventory7>
```

Java:

```
public int[] getInventory7(QueryData[] queryData)
```

Notice that "q" is defined as an index into the array.

The <xm:multiple> directive takes the following form:

<some-element xm:multiple="variable-type variable-name in parameter name">

The variable-type is the (optional) Java class name of the object that is bound in sequence to every component of the array. The variable-name is the name of the variable that iterates through the values in the array. It must be either a Java array or a class implementing the java.util.List or java.util.Collection interfaces.

Conclusion

In this article, we described the basic types of XML maps and the mapping directives available in WebLogic Workshop. In the next article, we'll show you how to use ECMAScripting.

Scripting capabilities are simply an extension of XML maps. They provide advanced mapping functions and are invaluable when the preprocessing requirements of your XML are complex.

Source Code

Download the Inventory.jws file here.

Post Comment