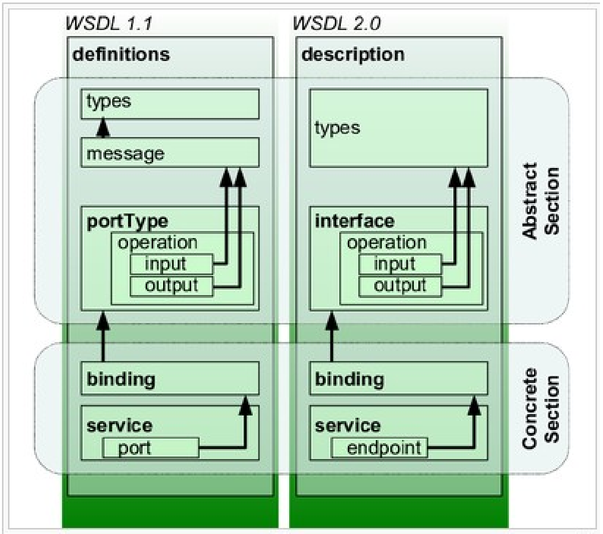
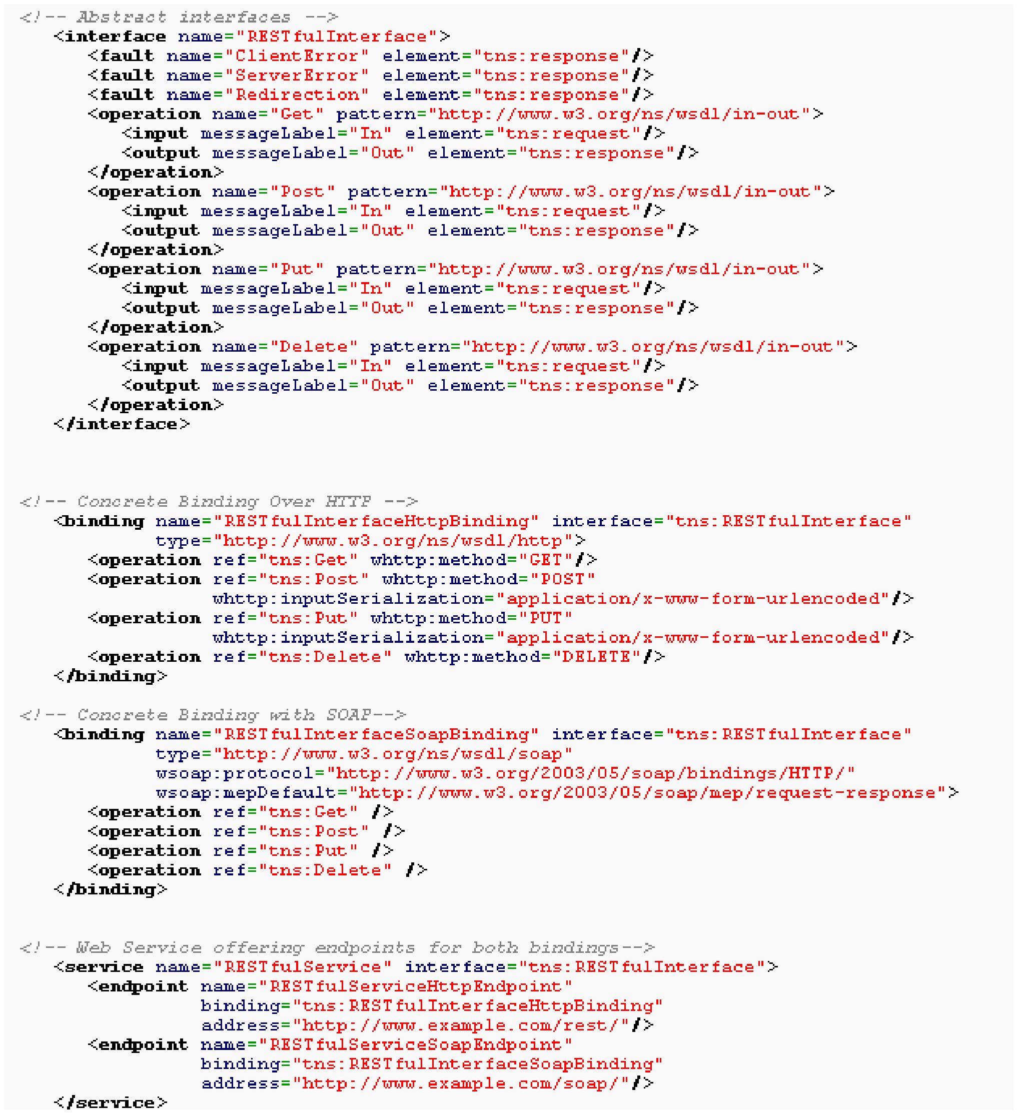
LIYANA ARACHCHIGE RANIL

WEB SERVICE RELATED SHORT NOTES

1. WSDL describes a web service
2. WSDL has two predominant version , 1.1 and 2.0 (2.0 is previously known as 1.2)
3. 

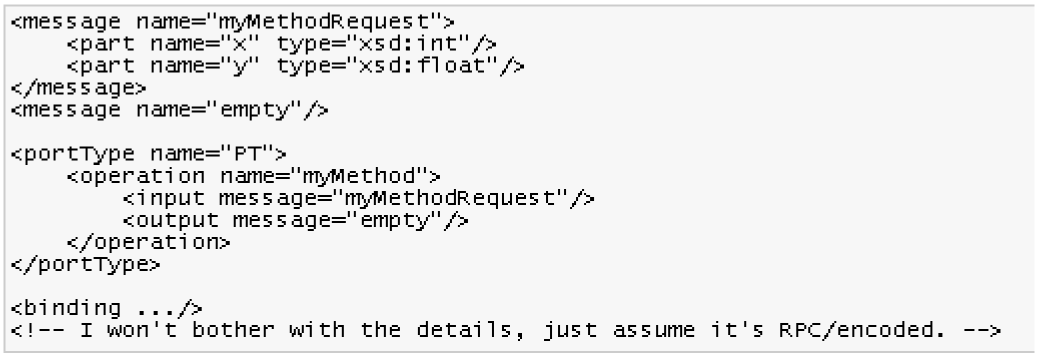
Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents.
4. WSDL binding describes a messaging protocol

5. 

```
<!-- Abstract interfaces -->
   <interface name="RESTfulInterface">
      <fault name="ClientError" element="tns:response"/>
      <fault name="ServerError" element="tns:response"/>
      <fault name="Redirection" element="tns:response"/>
      <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns:request"/>
         <output messageLabel="Out" element="tns:response"/>
      </operation>
      <operation name="Post" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns:request"/>
         <output messageLabel="Out" element="tns:response"/>
      </operation>
      <operation name="Put" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns:request"/>
         <output messageLabel="Out" element="tns:response"/>
      </operation>
      <operation name="Delete" pattern="http://www.w3.org/ns/wsdl/in-out">
         <input messageLabel="In" element="tns:request"/>
         <output messageLabel="Out" element="tns:response"/>
      </operation>
   </interface>


<!-- Concrete Binding Over HTTP -->
   <binding name="RESTfulInterfaceHttpBinding" interface="tns:RESTfulInterface"
            type="http://www.w3.org/ns/wsdl/http">
      <operation ref="tns:Get" whttp:method="GET"/>
      <operation ref="tns:Post" whttp:method="POST"
                 whttp:inputSerialization="application/x-www-form-urlencoded"/>
      <operation ref="tns:Put" whttp:method="PUT"
                 whttp:inputSerialization="application/x-www-form-urlencoded"/>
      <operation ref="tns:Delete" whttp:method="DELETE"/>
   </binding>

<!-- Concrete Binding with SOAP-->
   <binding name="RESTfulInterfaceSoapBinding" interface="tns:RESTfulInterface"
            type="http://www.w3.org/ns/wsdl/soap"
            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
            wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
      <operation ref="tns:Get" />
      <operation ref="tns:Post" />
      <operation ref="tns:Put" />
      <operation ref="tns:Delete" />
   </binding>


<!-- Web Service offering endpoints for both bindings-->
   <service name="RESTfulService" interface="tns:RESTfulInterface">
      <endpoint name="RESTfulServiceHttpEndpoint"
                binding="tns:RESTfulInterfaceHttpBinding"
                address="http://www.example.com/rest/"/>
      <endpoint name="RESTfulServiceSoapEndpoint"
                binding="tns:RESTfulInterfaceSoapBinding"
                address="http://www.example.com/soap/"/>
   </service>
```

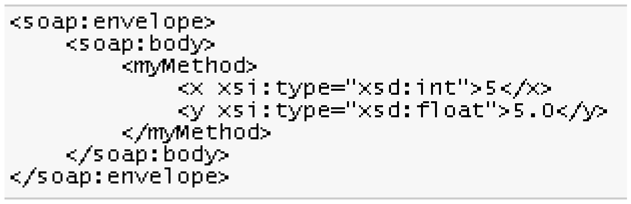
6. SOAP is such a messaging protocol which relies on HTTP transport mainly
7. WSDL binding could be RPC style or DOCUMENT style
8. RPC style would keep the name of the soap operation element in the SOAP massage while the DOCUMENT style would not keep the name of the SOAP operation.

Listing 2. RPC/encoded WSDL for myMethod



```
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
    <part name="y" type="xsd:float"/>
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's RPC/encoded. -->
```

9.



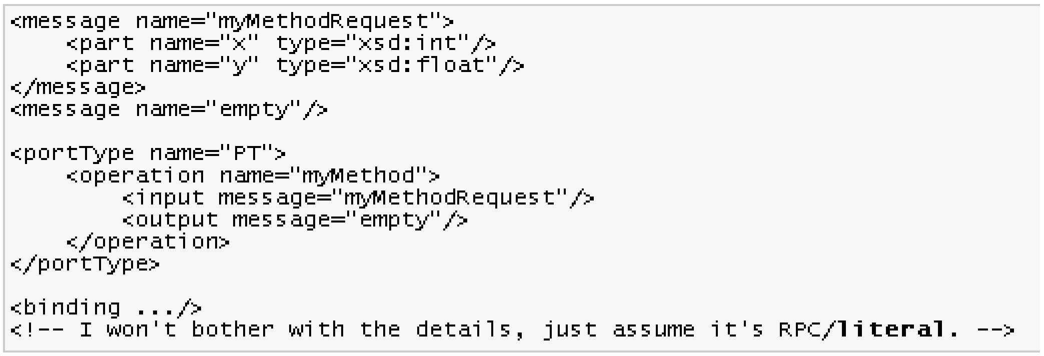
```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x xsi:type="xsd:int">5</x>
            <y xsi:type="xsd:float">5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

10.

11. RPC/LITERAL

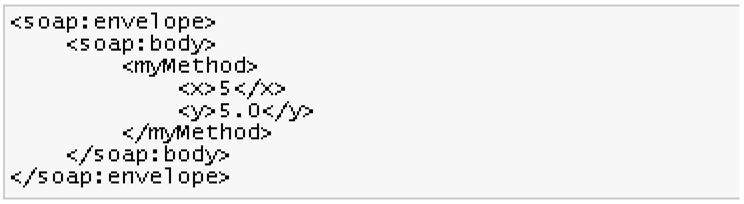
12.

Listing 4. RPC/literal WSDL for myMethod



```
<message name="myMethodRequest">
    <part name="x" type="xsd:int"/>
    <part name="y" type="xsd:float"/>
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's RPC/literal. -->
```

Listing 5. RPC/literal SOAP message for myMethod

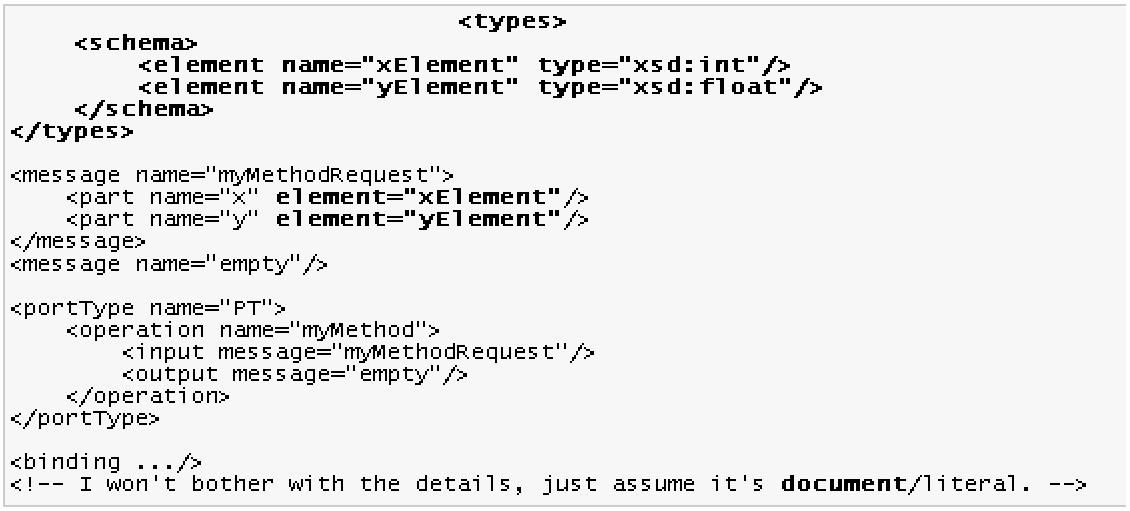


```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>5</x>
            <y>5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

13.

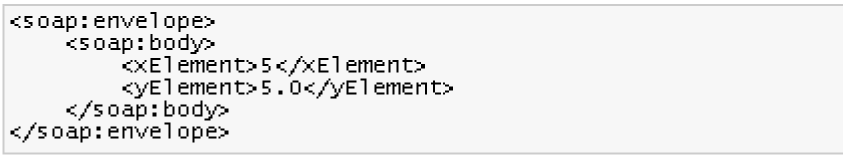
14. DOCUMENT / LITERAL

15.



```
                                  <types>
     <schema>
          <element name="xElement" type="xsd:int"/>
          <element name="yElement" type="xsd:float"/>
     </schema>
</types>

<message name="myMethodRequest">
    <part name="x" element="xElement"/>
    <part name="y" element="yElement"/>
</message>
<message name="empty"/>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's document/literal. -->
```

Listing 7. Document/literal SOAP message for myMethod



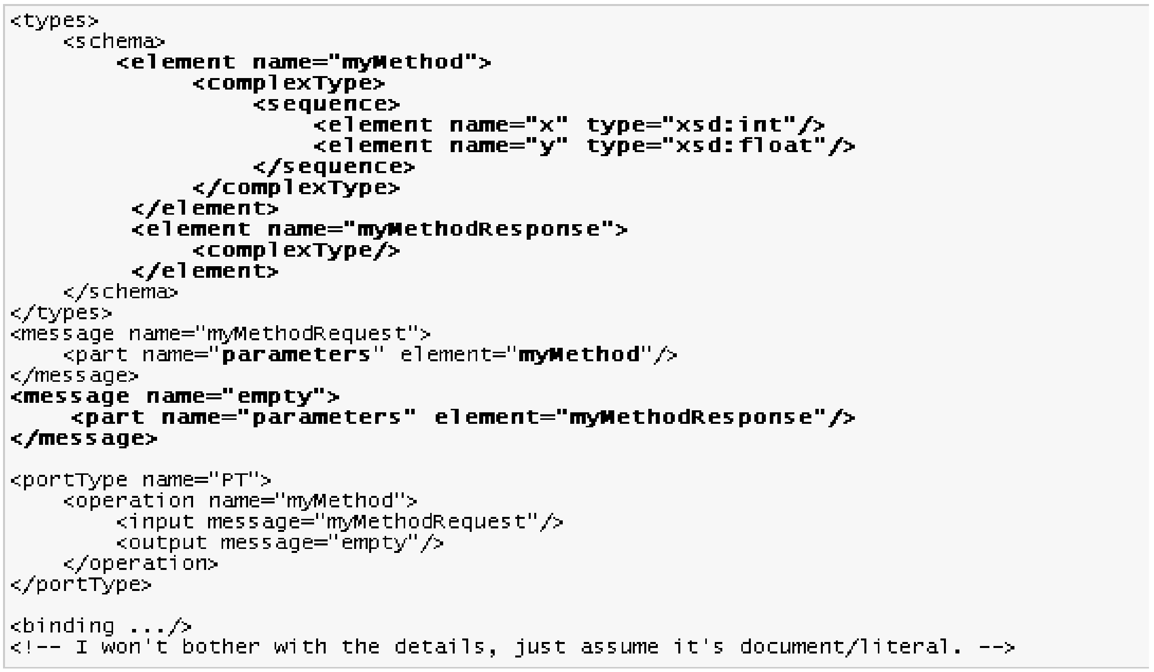
```
<soap:envelope>
    <soap:body>
        <xElement>5</xElement>
        <yElement>5.0</yElement>
    </soap:body>
</soap:envelope>
```

16.

17. To have the name of the operation use , DOCUMENT literal wrapped pattern

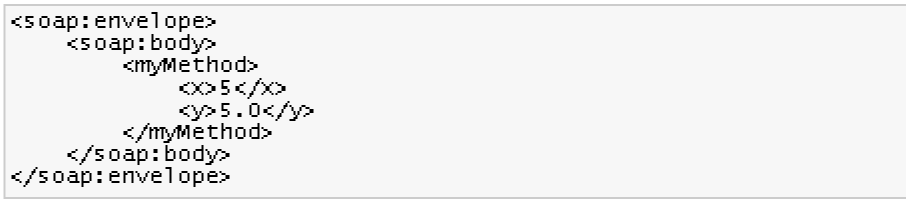
18.

Listing 8. Document/literal wrapped WSDL for myMethod



```
<types>
    <schema>
        <element name="myMethod">
             <complexType>
                 <sequence>
                     <element name="x" type="xsd:int"/>
                     <element name="y" type="xsd:float"/>
                 </sequence>
             </complexType>
         </element>
         <element name="myMethodResponse">
             <complexType/>
         </element>
    </schema>
</types>
<message name="myMethodRequest">
    <part name="parameters" element="myMethod"/>
</message>
<message name="empty">
     <part name="parameters" element="myMethodResponse"/>
</message>

<portType name="PT">
    <operation name="myMethod">
        <input message="myMethodRequest"/>
        <output message="empty"/>
    </operation>
</portType>

<binding .../>
<!-- I won't bother with the details, just assume it's document/literal. -->
```

Listing 9. Document/literal wrapped SOAP message for myMethod



```
<soap:envelope>
    <soap:body>
        <myMethod>
            <x>5</x>
            <y>5.0</y>
        </myMethod>
    </soap:body>
</soap:envelope>
```

19.

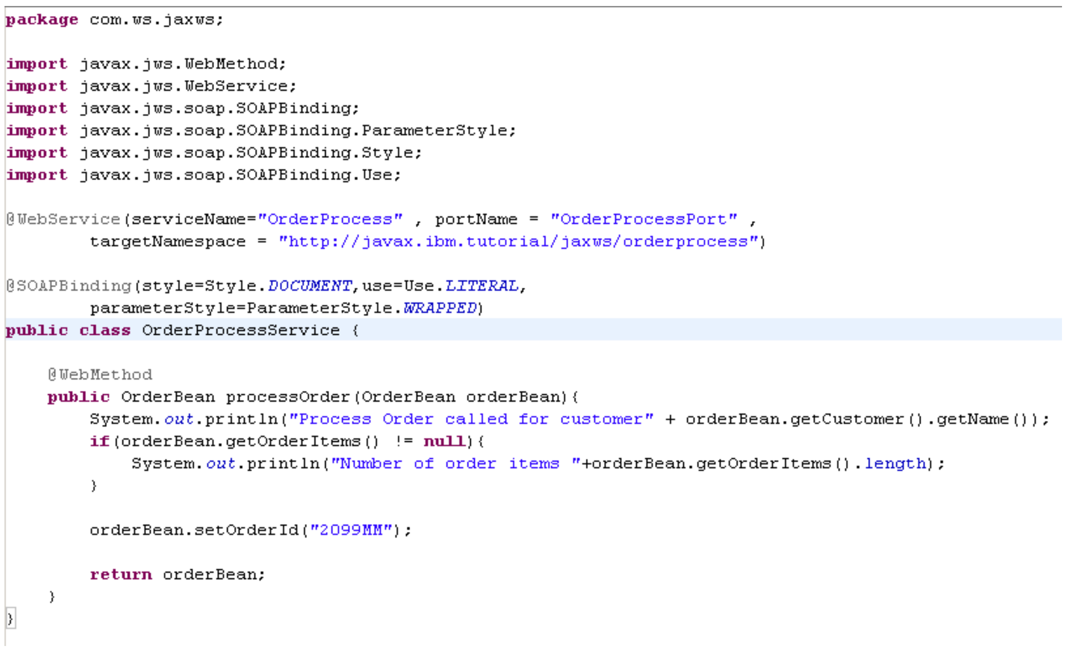
20. Both styles can use either SOAP encoding or not / ENCODED or LITERAL. Encoding will have additional data type encoding in SOAP messages , which is an additional overhead

21. **JAX-RPC 1.1** supports **WS-Interoperability Basic Profile 1.0.**

22. JAX-RPC 1.0 lacks support for WS-Interoperability Basic Profile 1.0 and this support is added in JAX-RPC 1.1
23. JAX-RPC 1.1 supports RPC/LITERAL , JAX-RPC 1.0 did not support this
24. **JAX-RPC 1.1** is required to support **RPC/ENCODED, RPC/LITERAL, DOCUMENT/LITERAL.** DOCUMENT/ENCODED is optional
25. WS- Interoperability Basic Profile 1.0 clarifies   SOAP 1.1 ,HTTP 1.1 , WSDL 1.1
26. WS – Interoperability Basic Profile 1.1 added additional things such as support for MTOM and WS-Addressing
27. JAX-RPC is by default RPC/ENCODED style
28. JAX-RPC is essentially JAVA RMI over SOAP
29. JAX-RPC has its own type mapping from XML->JAVA and JAVA->XML.
30. JAX-RPC type mapping is not fully complete in terms of XML schema types
31. JAX-RPC does not support WSDL binding to direct HTTP, it only supports WSDL binding to SOAP protocol only
32. JAX-RPC is targeted for J2EE1.4 platform. This has been replaced by JAX-WS in J2EE5
33. JAX-RPC handlers rely on SAAJ 1.2
34. JAX-WS handlers rely on SAAJ 1.3 which is compatible with both SAAJ 1.1 and 1.0
35. SAAJ 1.3 can handle SOAP 1.2 , SOAP 1.1 messages , while SAAJ 1.2 can only handle SOAP 1.1 messages
36. JAX-WS replaces JAX-RPC 1.1
37. JAX-WS still supports SOAP 1.1 , WSDL 1.1 and HTTP 1.1
38. JAX-WS also supports SOAP 1.2 , WSDL 2.0 (WSDSL 1.2)

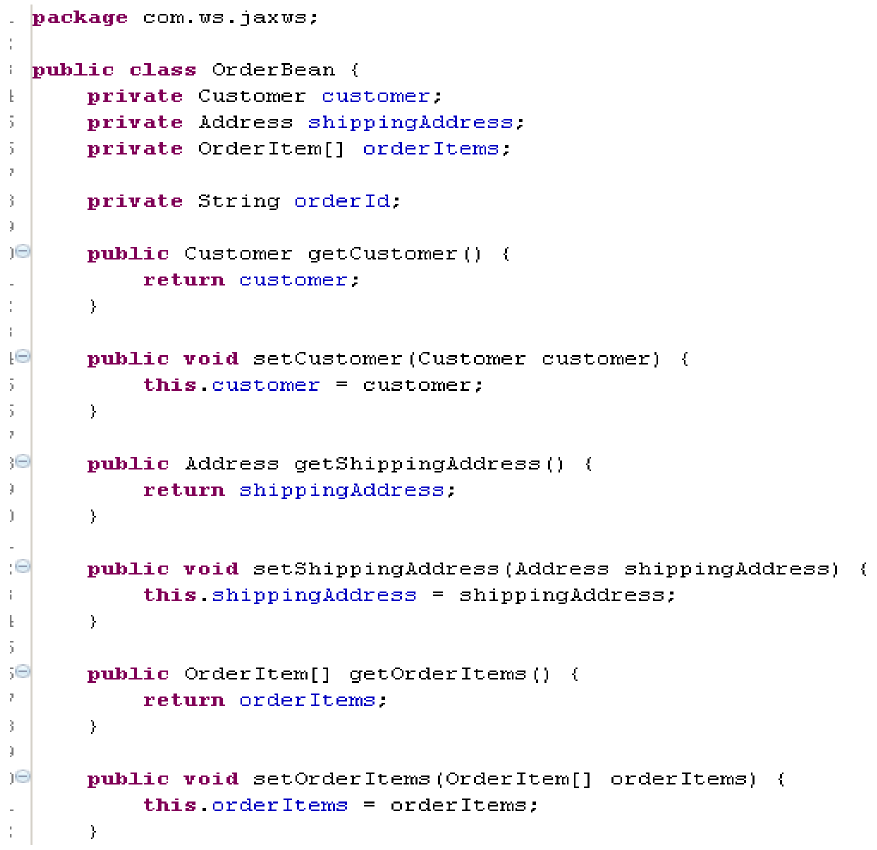
39. JAX-RPC session management was tied to HTTP , but JAX-WS provides message level SESSION MANAGEMENT
40. SERVICE REGISTRATION and DISCOVERY is not JAX-WS concern , its provided via another module called JAXR independently
41. JAX-WS is aimed at J2EE5 , while it use JAVA 5 , annotations, Not compatible with JAVA 1.4
42. JAX-WS supports binding , SOAP HTTP and XML HTTP both
43. JAX-WS supports WS-I Basic Profile 1.1
44. J2EE5 supports both JAX-WS and JAX-RPC
45. JAX-WS uses JAXB 2.0 as it’s data binding library
46. JX-WS only used JAXB as the binding library , you can not use external ones such as JIBX or CASTOR
47. JAX-WS can be used to generate Web services in both CONTRACT FIRST (Start WSDL contract and generate Java classes) and CODE FIRST (Start with a JAVA class and use annotation to generate both WSDL file and JAVA interface) approach
48. JAX-WS service implementation steps
    a. Create the Service Endpoint Implementation class (SEI) with @WebService and @WebMethod annotations

b.



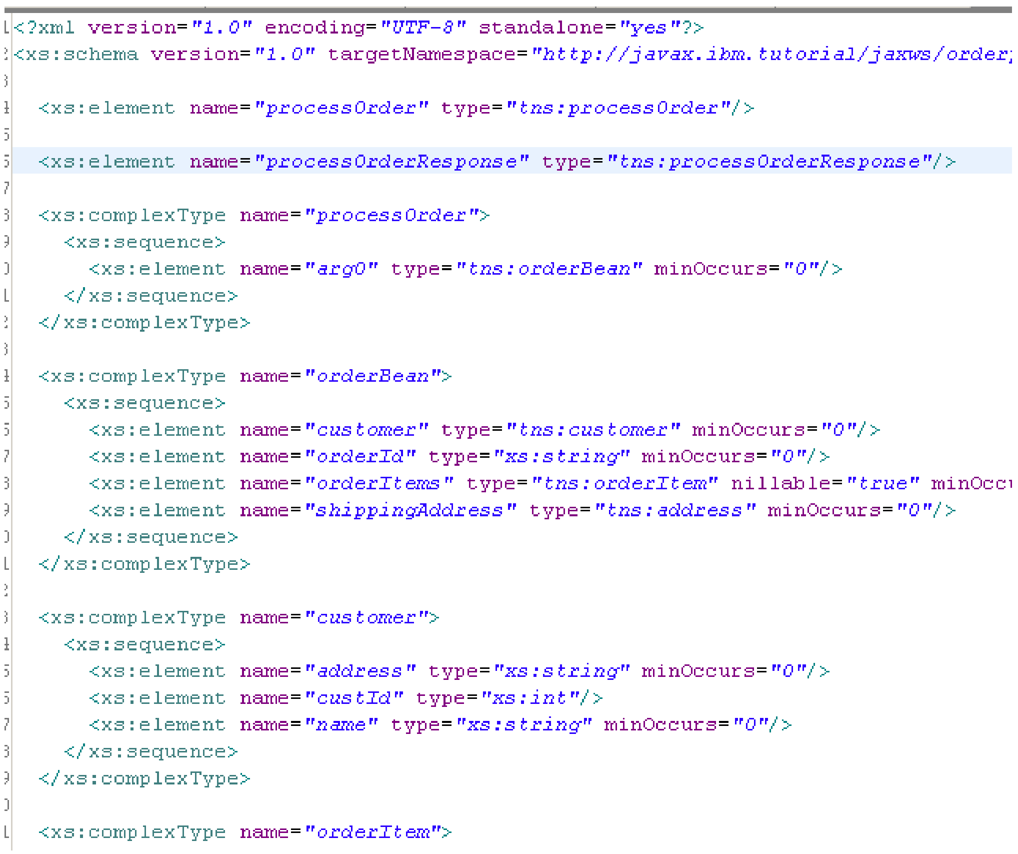
```
package com.ws.jaxws;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.ParameterStyle;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;

@WebService(serviceName="OrderProcess" , portName = "OrderProcessPort" ,
        targetNamespace = "http://javax.ibm.tutorial/jaxws/orderprocess")

@SOAPBinding(style=Style.DOCUMENT,use=Use.LITERAL,
        parameterStyle=ParameterStyle.WRAPPED)
public class OrderProcessService {

    @WebMethod
    public OrderBean processOrder(OrderBean orderBean){
        System.out.println("Process Order called for customer" + orderBean.getCustomer().getName());
        if(orderBean.getOrderItems() != null){
            System.out.println("Number of order items "+orderBean.getOrderItems().length);
        }

        orderBean.setOrderId("2099MM");

        return orderBean;
    }
}
```

c. Use WSGEN tool to generate any Java Bean bindings, WSDL and XSD for this SEI (**wsgen -cp . com.ibm.jaxws.tutorial.service.OrderProcessService -wsdl**). JAXB is in use when binding are generated d.



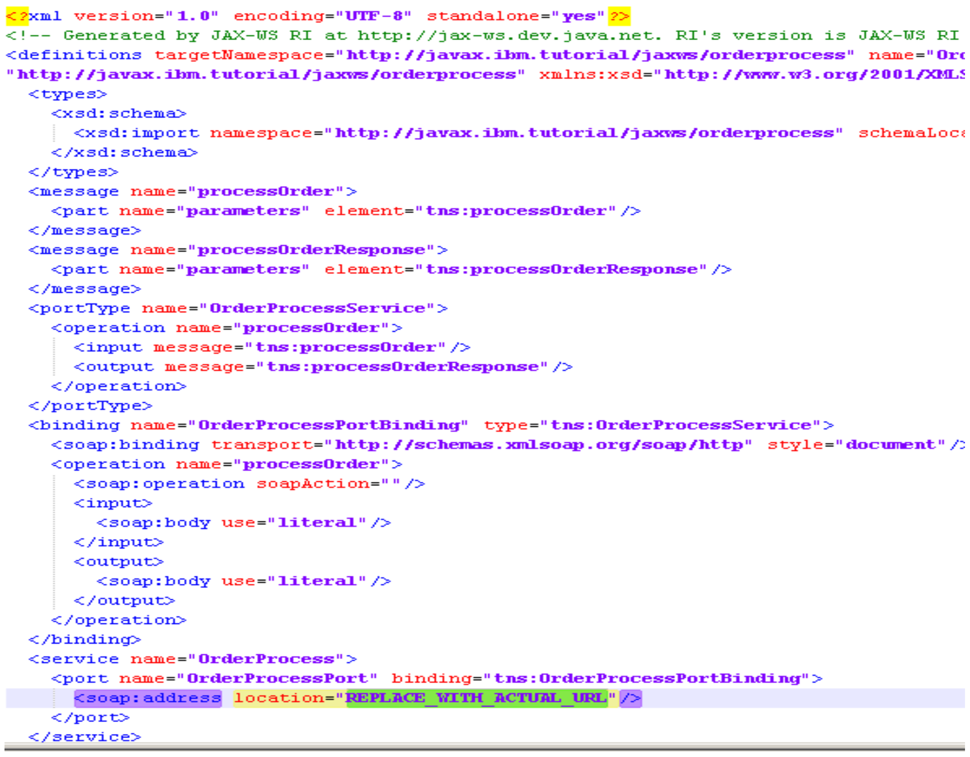
```
package com.ws.jaxws;

public class OrderBean {
    private Customer customer;
    private Address shippingAddress;
    private OrderItem[] orderItems;

    private String orderId;

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public Address getShippingAddress() {
        return shippingAddress;
    }

    public void setShippingAddress(Address shippingAddress) {
        this.shippingAddress = shippingAddress;
    }

    public OrderItem[] getOrderItems() {
        return orderItems;
    }

    public void setOrderItems(OrderItem[] orderItems) {
        this.orderItems = orderItems;
    }
```

e.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" targetNamespace="http://javax.ibm.tutorial/jaxws/order

  <xs:element name="processOrder" type="tns:processOrder"/>

  <xs:element name="processOrderResponse" type="tns:processOrderResponse"/>

  <xs:complexType name="processOrder">
    <xs:sequence>
      <xs:element name="arg0" type="tns:orderBean" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="orderBean">
    <xs:sequence>
      <xs:element name="customer" type="tns:customer" minOccurs="0"/>
      <xs:element name="orderId" type="xs:string" minOccurs="0"/>
      <xs:element name="orderItems" type="tns:orderItem" nillable="true" minOcc
      <xs:element name="shippingAddress" type="tns:address" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="address" type="xs:string" minOccurs="0"/>
      <xs:element name="custId" type="xs:int"/>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="orderItem">
```

f.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI
<definitions targetNamespace="http://javax.ibm.tutorial/jaxws/orderprocess" name="Or
"http://javax.ibm.tutorial/jaxws/orderprocess" xmlns:xsd="http://www.w3.org/2001/XML
  <types>
    <xsd:schema>
      <xsd:import namespace="http://javax.ibm.tutorial/jaxws/orderprocess" schemaLoc
    </xsd:schema>
  </types>
  <message name="processOrder">
    <part name="parameters" element="tns:processOrder" />
  </message>
  <message name="processOrderResponse">
    <part name="parameters" element="tns:processOrderResponse" />
  </message>
  <portType name="OrderProcessService">
    <operation name="processOrder">
      <input message="tns:processOrder" />
      <output message="tns:processOrderResponse" />
    </operation>
  </portType>
  <binding name="OrderProcessPortBinding" type="tns:OrderProcessService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" /
    <operation name="processOrder">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <service name="OrderProcess">
    <port name="OrderProcessPort" binding="tns:OrderProcessPortBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL" />
    </port>
  </service>
```

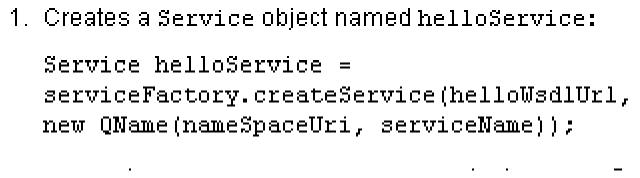
g. After generating all the ARTIFACTS , deploy the JAX-WS web service endpoint in a respective JAX-WS supported server

49. JAXB2.0 supports for all XML schema data types
50. JAX-WS supports asynchronous functionality (by means of Call back or Polling) in it's interfaces which is lacking in JAX-RPC
51. JAX-WS supports MTOM (Message Transmission Optimization Mechanism) , JAX-RPC does not support this
52. JAX-WS handlers rely on SAAJ 1.3

53. JAX-WS supports only DOCUMENT literal approach , SOAP encoding is not supported
54. WS-I Basic Profile provides guidance to use SOAP 1.1 , WSDL 1.1 and UDDI 2.0
55. Apache AXIS 1.x is an implementation of JAX-RPC
56. Apache AXIS 1.x relies on SAX to process messages
57. Apache AXIS 2 is not JAX-RPC , it is a proprietary implementation
58. Apache AXIS 2 is designed on AXIOM (AXIS Object Model)
59. AXIOM relies on StAX for XML processing
60. AXIS2 has JAX-WS API support
61. Apache CXF is a combination of Celtix and XFire
62. Apache CXF is a Web service framework
63. CXF has JAX-WS API support
64. CXF support JAX-RS (API for RESTful web services)
65. There are three different types of web service clients (JAX-RPC) are possible to be created. Dynamic Proxy (Need the WSDL URL to create proxies, good for situations where WDL is very dynamic which changes frequently) , Dynamic Invocation Interface (DII, no need anything other than the ENDPOING address, very dynamic) , Static Stub Clients ,Application Client (Via JNDI lookups)
66. Dynamic Proxy Client in JAX-RPC 1.1 is not suitable for web service invocations where complex types are used. It is recommended to use Dynamic Proxy in cases where the web service method have only Java primitive types
67. Dynamic Proxy , Client Stub would always need web service endpoint developers to give a way required classes , in case of Dynamic Proxy , at least Interface definition (Can be taken from

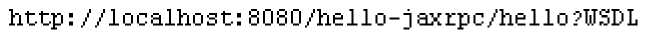
WSDL URL as well) , in case of Client Stub all the stub related file generated using wscompile tool.

68. Also WSCOMPILE tool can be used to generate DYNAMIC PROXY interface for Dynamic Proxy Clients

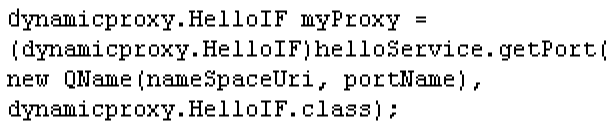
69. 

1. Creates a Service object named helloService:

```
Service helloService =
serviceFactory.createService(helloWsdlUrl,
new QName(nameSpaceUri, serviceName));
```

70. 

```
http://localhost:8080/hello-jaxrpc/hello?WSDL
```

71. 

```
dynamicproxy.HelloIF myProxy =
(dynamicproxy.HelloIF)helloService.getPort(
new QName(nameSpaceUri, portName),
dynamicproxy.HelloIF.class);
```

72. The amount of code required to do in case of Client Stub is lesser that Dynamic Proxy and Dynamic Invocation Interface (DII)

73. In case of DII , service implementers does not have to give anything other than the ENDPOING ADDRESS

74. DII lets programmers to interact with SOAP XML request and response messages

75. You can use SAAJ for sending SOAP messages as a NON-RPC option as well

76. Dynamic Proxy and DII are believed to be slower when compared to Client Stubs

77. JAX-WS approach is similar to JAX-RPC though there is less code involved

78. JAX-WS requires lot of J2SE 5 annotation features

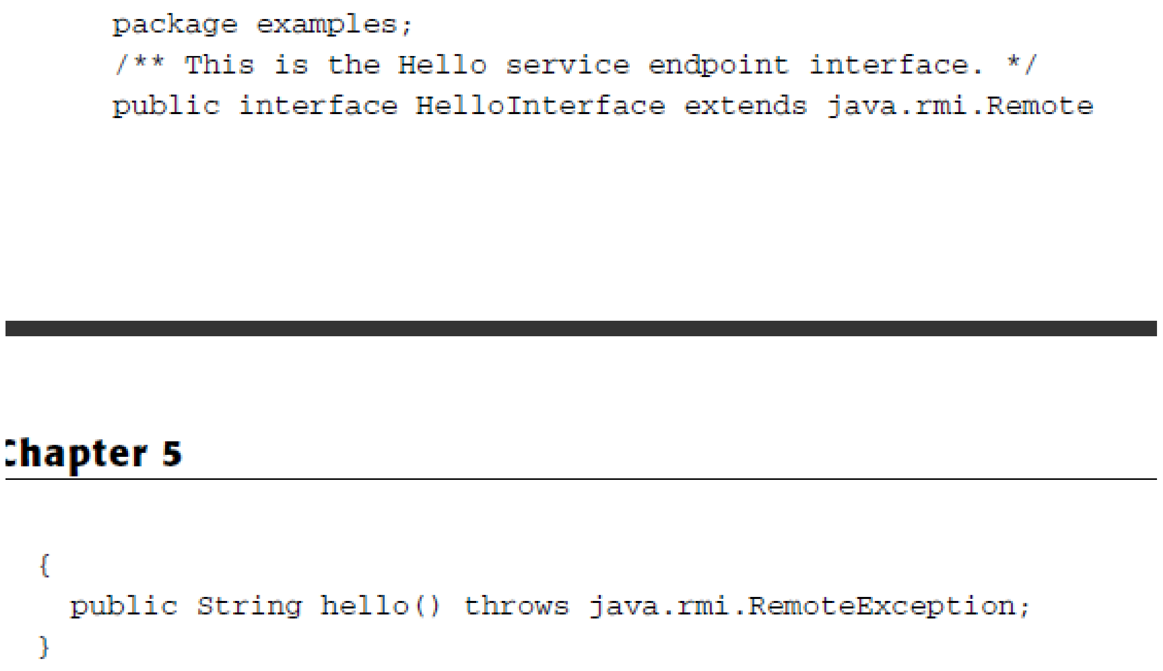
79. **JAX-WS clients** can be written using **Dynamic Dispatch Client API** (Dynamic Client programming model), and the **Dynamic Proxy Client API** (static client programming model)

80. Both Dynamic Dispatch Client API and Dynamic Proxy Client API supports both SYNCHRONUS and ASYNCHRONOUS invocations

81. Dynamic Dispatch client API is useful when you want to work with XML message level without any generated Artifacts at the JAX-WS level
82. Dynamic Dispatch Client can send data in either PAYLOAD or MESSAGE mode
83. Dynamic Dispatch Clients supports ,**javax.xml.transform.Source** , **JAXB** Objects **or javax.activation.DataSource**
84. Dynamic Proxy Client – Use this when you want to invoke a Web Service based on an END POINT interface
85. Dynamic Proxy Client is the equivalent in Stub Clients in JAX-RPC
86. But in JAX-WS the Dynamic Proxy Client does not need static classes as in the case of JAX-RPC. The proxy classes are generated at the RUN TIME in case of JAX-WS using WSDL
87. Dynamic Dispatch Client provides more flexibility than JAX-RPC Dynamic Invocation Interface (DII)
88. JAXB is used for JAVA-to-XML and XML-to-JAVA binding. Java-to-XML binding is known as Marshalling while XML-to-Java is know as Un marshalling
89. JAXB is used to convert a .XSD to corresponding mapped JAVA classes and also to create .XSD from Java Classes
90. EJB3.0 / EJB2.1 supports converting a STATELESS SESSION BEAN to a WEB SERVICE END POINT
91. EJB 2.0 DID NOT have support for WEB SERVICES
92. EJB 2.1 Web service implementation
    a. Create an Additional REMOTE interface for STATELESS SESSION bean which is going to be EXPOSED as a WEB SERVICE END POINT

b. The SESSION BEAN DOES NOT NEED to implement this REMOTE interface, just REMOTE interface needs to have same METHOD SIGNATURES of the METHODS that are going to be EXPOSED

c.



```
package examples;
/** This is the Hello service endpoint interface. */
public interface HelloInterface extends java.rmi.Remote
```

hapter 5

```
{
  public String hello() throws java.rmi.RemoteException;
}
```

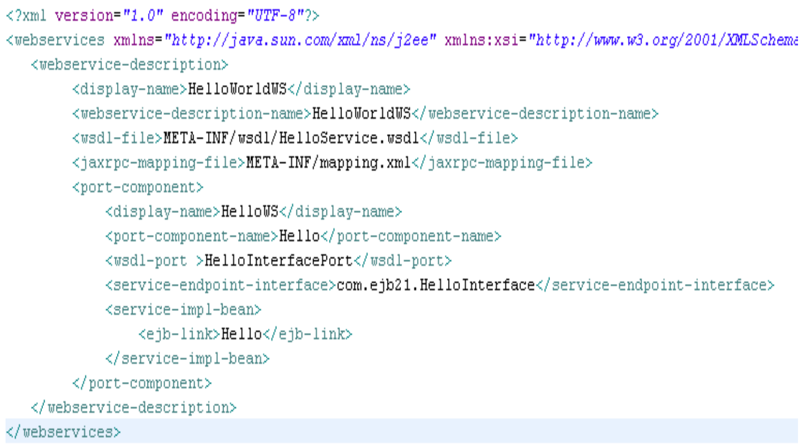
d. The interface MUST extend java.rmi.Remote interface

e. All methods MUST throw REMOTE EXCEPTION

f. The method prams , return types MUST be JAVA types SUPPORTED by JAX-RPC

g. Service Endpoint Interface MUST NOT include CONSTANTS

h. In addition to that the CONTAINER needs to know to which Implementation class that incoming SOAP messages should be dispatched. For this “**webservices.xml**” file is provided in META-INF



```
<?xml version="1.0" encoding="UTF-8"?>
<webservices xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchem
   <webservice-description>
        <display-name>HelloWorldWS</display-name>
        <webservice-description-name>HelloWorldWS</webservice-description-name>
        <wsdl-file>META-INF/wsdl/HelloService.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/mapping.xml</jaxrpc-mapping-file>
        <port-component>
            <display-name>HelloWS</display-name>
            <port-component-name>Hello</port-component-name>
            <wsdl-port >HelloInterfacePort</wsdl-port>
            <service-endpoint-interface>com.ejb21.HelloInterface</service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>Hello</ejb-link>
            </service-impl-bean>
        </port-component>
   </webservice-description>
</webservices>
```

i.

j. In addition to that the corresponding WSDL must also be provided , to GENERATE the WSDL the CONTAINER provider usually provides TOOLS (such as wscompile)

k. Webservice.xml file would specify where to find the WSDL and JAX-RPC mapping file

93. EJB 3.0 Web Service Implementation

a. The CLASS must be annotated with **@WebSerivice** or **@WebServiceProvider**

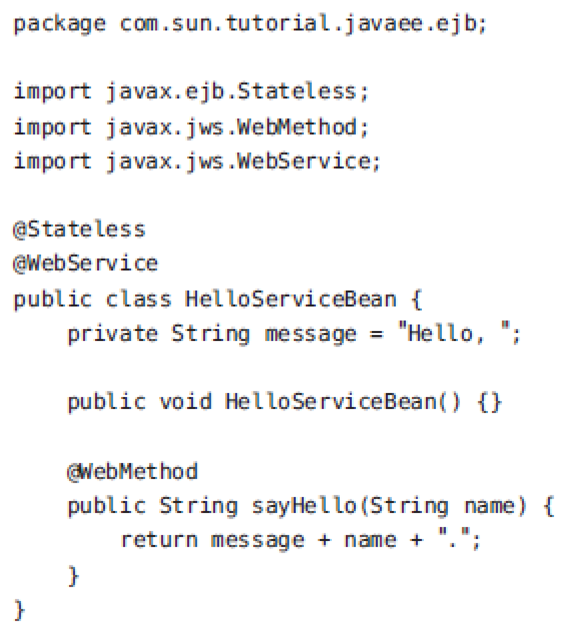
b. The implementing CLASS may reference an END POINT INTERFACE using “endpointInterface” element. But this is NOT COMPULSORY

c. If no “enpointInterface” is defined , the interface is IMPLICITLY DEFINED for implementing CLASS

d. Business Methods MUST BE , **PUBLIC** , **MUST NOT** be **STATIC** or **FINAL**

e. Business Methods that are exposed to WEB SERVICE CLIENTS MUST be annotated with **@WebMethod**

f. Business methods exposed must have arguments , return types COMPATIBLE to JAXB
g. Implementing class **MUST NOT** be **FINAL** or **ABSTRACT**
h. Implementing class MUST HAVE a DEFAULT PUBLIC CONSTRUCTOR
i. The ENDPOINT class MUST BE annotated with **@Stateless**
j. The Implementing class may use **@PreDestroy** or **@PostConstruct**



```
package com.sun.tutorial.javaee.ejb;

import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService
public class HelloServiceBean {
    private String message = "Hello, ";

    public void HelloServiceBean() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

k.

94. EJB based WEB SERVICES are managed by the CONTATINER
95. EJB 3.0 web services are heavily annotation based

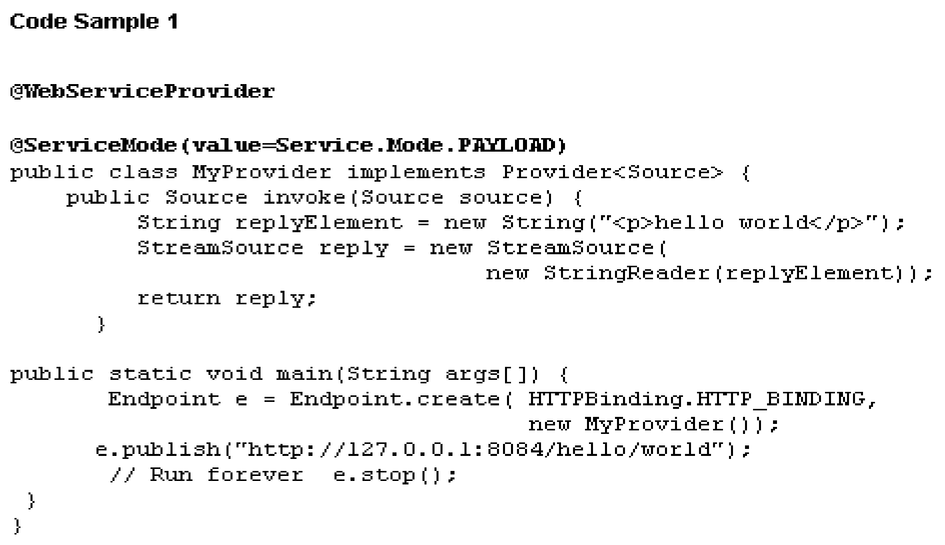
RESTFUL WEB SERVICES

1. RESTful web service stands for REPRESENTATIONAL STATE TRANSFER

2. In RESTful web services , the WS is viewed as RESOURCES while identified by their URL(S)
3. The HTTP methods such as GET , POST are the VERBS that the developer can use to describe the necessary CREATE , READ, UPDATE and DELETE (CRUD) actions to be performed
4. However REST style and HTTP protocol are MUSTUALLY EXCLUSIVE , the REST DOES NOT REQUIRE HTTP
5. RESTful Web Services / When to Use
    a. Web Services are Completely STATELESS
    b. The SERVICE PRODUCER and SERVICE CONSUMER have a MUTUAL understanding of the CONTEXT and CONTENT being passed along. Because there is no FORMAL way to DESCRIBE the Web Service INTERFACEC
    c. REST is particularly USEFUL for LIMITED PROFILE devices such as PDA , Mobile Phones
6. SOAP based web services / When to Use
    d. A FORMAL contract must be established to describe the interfaces that the Web Service offer
    e. Architecture must address complex NON-FUNCTIONALI requirements such as TRANSACTIONS, SECURITY, ADDRESSING, TRUST, COORDINATION etc. Most real world application go BEYOND simple CRUD operations and require CONTEXTUAL information and CONVERSATIONAL STATE to be MAINTAINED
    f. The Architecture needs to handle ASYNCHRONOUS processing and INVOCATION
7. JAX-WS provides FULL support for RESTful web services
8. JAX-WS provides building RESTful endpoints through a javax.xml.ws.Provider interface. Provider is a GENERIC interface

that can be implemented by a CLASS as a dynamic alternative to a SERVICE END POING INTERFACE

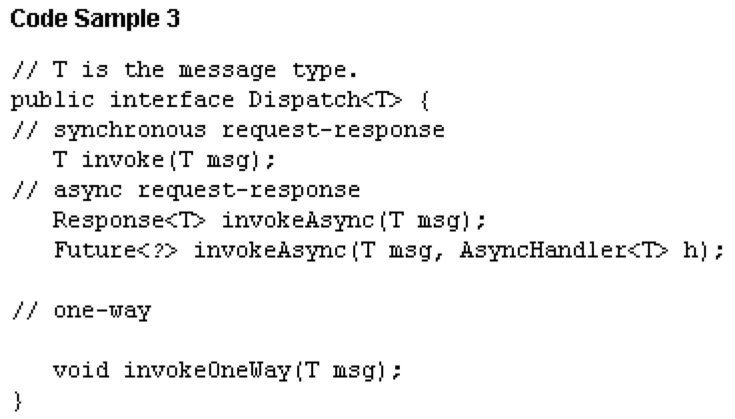
9. Service implementing **Provider** interface can be DEPLOYED in a J2EE container or published in a STAND-ALONE mode through the JAX-WS Endpoing API
10.



Code Sample 1

```
@WebServiceProvider

@ServiceMode(value=Service.Mode.PAYLOAD)
public class MyProvider implements Provider<Source> {
    public Source invoke(Source source) {
         String replyElement = new String("<p>hello world</p>");
         StreamSource reply = new StreamSource(
                              new StringReader(replyElement));
         return reply;
      }

public static void main(String args[]) {
       Endpoint e = Endpoint.create( HTTPBinding.HTTP_BINDING,
                                 new MyProvider());
      e.publish("http://127.0.0.1:8084/hello/world");
       // Run forever  e.stop();
 }
}
```

11. Application can access RESTful web services in two ways , from the BROWSER or PROGRAMMATICALLY
12. In JAX-WS , **use javax.xml.ws.Dispatch** for accessing RESTful web services programmatically



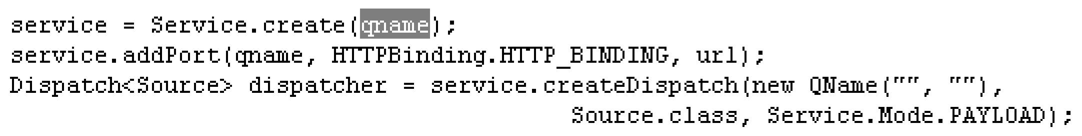
Code Sample 3

```
// T is the message type.
public interface Dispatch<T> {
// synchronous request-response
   T invoke(T msg);
// async request-response
   Response<T> invokeAsync(T msg);
   Future<?> invokeAsync(T msg, AsyncHandler<T> h);

// one-way

   void invokeOneWay(T msg);
}
```

13.

14. Unlike the PROVIDER on Server side , developers do not implement this API (Dispatch) , instead they obtain an instance from the Service object

15.



```
service = Service.create(qname);
service.addPort(qname, HTTPBinding.HTTP_BINDING, url);
Dispatch<Source> dispatcher = service.createDispatch(new QName("", ""),
                                        Source.class, Service.Mode.PAYLOAD);
```

16. @WebServiceProvider vs @WebService , @WebServiceProvider is used with Provider interface implementations and there is only one method in the interface. @WebService is for SEI which could have more than one methods annotated with @WebMethod