JAVA BEANS:

JAVABEANS CLASSES

- JSTL Core actions are designed to be used for simple, presentation-oriented programming tasks
- More sophisticated programming tasks should still be performed with a language such as Java
- JavaBeans technology allows a JSP document to call Java methods

Requirements:

JavaBeans class must

- Be public and not abstract
- Contain at least one *simple property design pattern* method .
    - Simple property design patterns
        - Two types: getter and setter
            - Both require that the method be public
            - getter:
                - no arguments
                - returns a value (i.e., cannot be void)
                - name begins with get (or is, if return type is boolean) followed by upper case letter
            - setter:
                - one argument (same type as getter return value)
                - Void (i.e., must not return a value)
                - name begins with set followed by upper case letter
- Class must have a default (no-argument) constructor to be instantiated by useBean .
- Class should belong to a package

StudentsBean.java:

```
package com;
```

```
public class StudentsBean implements java.io.Serializable
{
  private String firstName = null;
  private String lastName = null;
  private int age = 0;

  public StudentsBean() {
  }
  public String getFirstName(){
    return firstName;
  }
  public String getLastName(){
    return lastName;
  }
  public int getAge(){
    return age;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setAge(int age){
    this.age = age;
  }
}
```

Jsp_Bean.jsp:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
      <title>JSP Page</title>
   </head>
   <body>
<jsp:useBean id="students" class="com.StudentsBean">
  <jsp:setProperty name="students" property="firstName"
                value="Zara"/>
  <jsp:setProperty name="students" property="lastName"
                value="Ali"/>
  <jsp:setProperty name="students" property="age"
                value="10"/>

</jsp:useBean>

<p>Student First Name:
  <jsp:getProperty name="students" property="firstName"/>
</p>
<p>Student Last Name:
  <jsp:getProperty name="students" property="lastName"/>
</p>
<p>Student Last Name:
  <jsp:getProperty name="students" property="age"/>
</p>
</body>
</html>
```

XML DOCUMENTS:

- An XML document is one that follows certain syntax rules (most of which we followed for XHTML)

- An XML document consists of
  - Markup
    - Tags, which begin with < and end with >

        - References, which begin with & and end with ;
            - Character, e.g.
            - Entity, e.g. <
    - Character data: everything not markup

    Staff1.xml:

```
<?xml version="1.0"?>
<company>
        <staff id="1001">
                <firstname>yong</firstname>
                <lastname>mook kim</lastname>
                <nickname>mkyong</nickname>
                <salary>100000</salary>
        </staff>
        <staff id="2001">
                <firstname>low</firstname>
                <lastname>yin fong</lastname>
                <nickname>fong fong</nickname>
                <salary>200000</salary>
        </staff>
</company>
```

- Element tags and elements
    - Three types
        - Start, e.g. <message>
        - End, e.g. </message>
        - Empty element, e.g.
- XML Rules:
    - Start and end tags must properly nest
    - Corresponding pair of start and end element tags plus everything in between them defines an element
    - Character data may only appear within an element

- Start and empty-element tags may contain attribute specifications separated by white space
  - Syntax: *name = quoted value*

Well-formed XML document:

- A well-formed XML document
  - follows the XML syntax rules and
  - has a single root element
- Well-formed documents have a tree structure
- Many XML parsers (software for reading/writing XML documents) use tree representation internally

XML parsers

- Two types of XML parsers:
  - Validating
    - Requires document type declaration
    - Generates error if document does not
      - Conform with DTD and
      - Meet XML validity constraints
        - » Example: every attribute value of type ID must be unique within the document
  - Non-validating
    - Checks for well-formedness
    - Can ignore external DTD

XML NAMESPACES:

- XML Namespace: Collection of element and attribute names associated with an XML vocabulary (such as XHTML)
- Namespace Name: Absolute URI that is the name of the namespace
  - Ex: http://www.w3.org/1999/xhtml is the namespace name of XHTML 1.0
- Default namespace for elements of a document is specified using a form of the xmlns attribute:

Another form of xmlns attribute known as a namespace declaration can be used to associate a namespace prefix with a namespace name:

xmlns:h="http://www.w3.org/TR/html4/"

- In **a namespace-aware** XML application, all element and attribute names are considered qualified names
  - A qualified name has an associated expanded name that consists of a namespace name and a local name
  - Ex: <table> is a qualified name with expanded name <null, table>
  - Ex: <h:table> is a qualified name with expanded name
    < http://www.w3.org/TR/html4, table>

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications. This XML carries HTML table information:

<table>
<tr>
<td>Apples</td>
<td>Bananas</td>
</tr>
</table>

This XML carries information about a table (a piece of furniture):

<table>
<name>African Coffee Table</name>
<width>80</width>
<length>120</length>
</table>

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.An XML parser will not know how to handle these differences.

Solving the Name Conflict Using a Prefix

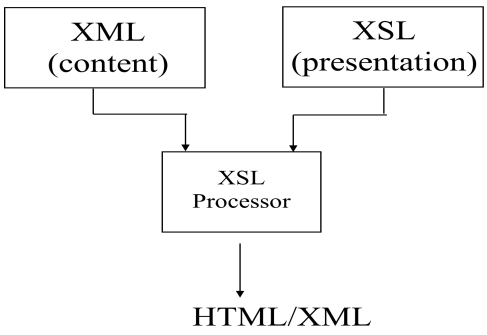
When using prefixes in XML, a so-called **namespace** for the prefix must be defined. The namespace is defined by the **xmlns attribute** in the start tag of an element. The namespace declaration has the following syntax.

xmlns:*prefix*="*URI*".

```
<root>
<h:table xmlns:h="http://www.w3.org/TR/html4/">
<h:tr> <h:td>Apples</h:td> <h:td>Bananas</h:td> </h:tr>
</h:table>
<f:table xmlns:f="http://www.w3schools.com/furniture">
<f:name>African Coffee Table</f:name>
<f:width>80</f:width>
<f:length>120</f:length>
</f:table>
</root>
```

XSL:

- XSL = eXtensible Stylesheet Language
- XSL consists of
  - XPath (navigation in documents)
  - XSLT (T for *transformations)*
  - XSLFO (FO for *formatting objects)*
    - This is a rather complex language for typesetting (i.e., preparing text for printing)



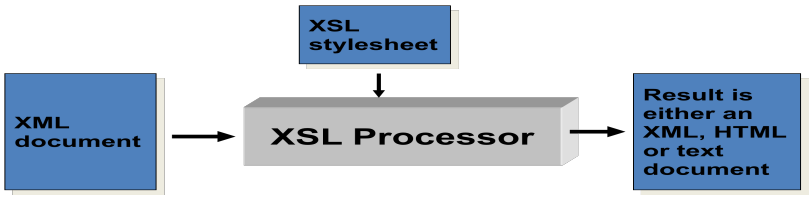
- An XSL stylesheet is a program that transforms an XML document into another XML document
- For example:
  - Transforming XML to XHTML (HTML that conforms to XML syntax)
  - Transforming an XML document to WML (a format of XML that cellular phones can display)

Applying XSLT Stylesheets to XML Documents:

- Adding to the XML document a link to the XSL stylesheet and letting the browser do the transformation

  <?xml-stylesheet type="text/xsl" href="trans.xsl"?>

- **Using an XSL Processor:**



trans.xml:

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

<?xml-stylesheet type="text/xsl" href="trans.xsl"?>

<country>
```

```
<language>
<name>Kannada</name>
<region>Karnataka</region>
<users>38M</users>
<family>Dravidian</family>
<style>kannada style </style>
</language>
<language>
<name>Tamil</name>
<region>Tamil Nadu</region>
<users>40M</users>
<family>Dravidian</family>
<style>kannada style </style>
</language>
</country>
```

trans.xsd:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h1>Indian Languages details</h1>
        <table border="1">
          <tr>
```

```
                <th>Language</th>
                <th>Family/Origin</th>
                <th>No. of speakers</th>
                <th>Region</th>
            </tr>
        <xsl:for-each select="country/language">
            <tr>
                <td><xsl:value-of select="name"/></td>
                <td><xsl:value-of select="family"/></td>
                <td><xsl:value-of select="users"/></td>
                <td><xsl:value-of select="region"/></td>
            </tr>
          </xsl:for-each>
         </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Indian Languages details

Language	Family/Origin	No. of speakers	Region
Kannada	Dravidian	38M	Karnataka
Tamil	Dravidian	40M	Tamil Nadu

CREATING A JAVA WEB SERVICE:

1. Create a java web application
2. Right click the project create java web service and give package name as pac1 and web service name as Calculator

3. Include a web method named “add “ by Calculator->right click->Add Operation.
4. Include a web method named “sub “ by Calculator->right click->Add Operation.
5. Include
    a. I/P PARAM : int a,int b
    b. O/P PARAM : return ()
6. Clean and build the project.
7. Deploy the project.
8. Test the service by Calculator->right click->Test Web Service.
9. Copy the WSDL URL.

CODE:

```
package p1;

import javax.jws.WebService;

import javax.jws.WebMethod;

import javax.jws.WebParam;

@WebService(serviceName = "Calculator")

public class Calculator

{

   @WebMethod(operationName = "add")

   public int add(@WebParam(name = "a") int a, @WebParam(name = "b") int b)

   {

      return a+b;

   }

   @WebMethod(operationName = "sub")

   public int sub(@WebParam(name = "a") int a, @WebParam(name = "b") int b)

  {

      return a-b;

   }

}
```

Calculator.WSDL:

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://p1/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://p1/"
name="Calculator">

<types>
<xsd:schema>
<xsd:import namespace="http://p1/"
schemaLocation="http://localhost:8080/Calc/Calculator?xsd=1"/>
</xsd:schema>
</types>

<message name="add">
<part name="parameters" element="tns:add"/>
</message>

<message name="addResponse">
<part name="parameters" element="tns:addResponse"/>
</message>

<message name="sub">
<part name="parameters" element="tns:sub"/>
</message>

<message name="subResponse">
<part name="parameters" element="tns:subResponse"/>
  </message>

<portType name="Calculator">
<operation name="add">
<input wsam:Action="http://p1/Calculator/addRequest" message="tns:add"/>
<output wsam:Action="http://p1/Calculator/addResponse" message="tns:addResponse"/>
</operation>
<operation name="sub">
<input wsam:Action="http://p1/Calculator/subRequest" message="tns:sub"/>
<output wsam:Action="http://p1/Calculator/subResponse" message="tns:subResponse"/>
</operation>
</portType>
```

```
<binding name="CalculatorPortBinding" type="tns:Calculator">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="add">
<soap:operation soapAction=""/>
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
<operation name="sub">
<soap:operation soapAction=""/>
<input>
  <soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>

<service name="Calculator">
<port name="CalculatorPort" binding="tns:CalculatorPortBinding">
<soap:address location="http://localhost:8080/Calc/Calculator"/>
</port>
</service>

</definitions>
```

Analysis of the Example
1. Definition : ***Calculator***

2. Type : Using built-in data types and they are defined in XMLSchema.

3. Message :

- **addRequest** : parameter a,b
- **addresponse**: return a+b
- **subRequest** : parameter a,b

- **subresponse**: return a-b

4. Port Type: ***add*** operation that consists of a request and response service.

5. Binding: Direction to use the SOAP HTTP transport protocol.

6. Service: Service available at ***http://localhost:8080/Service2/Calculator***

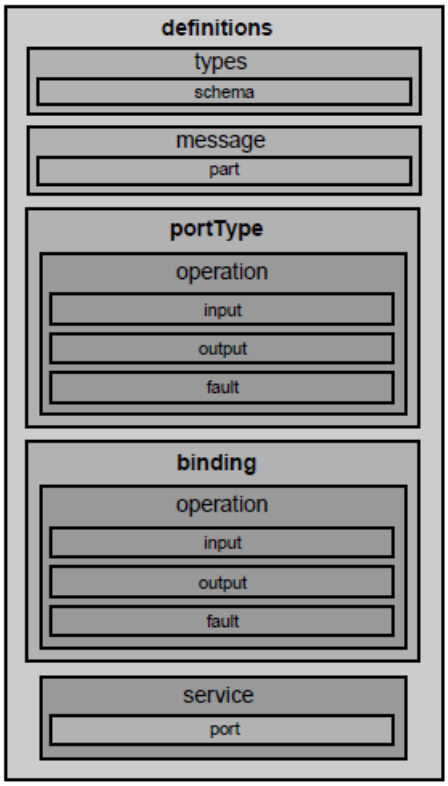
7. Port: Associates the binding with the URI ***http://localhost:8080/Service2/Calculator*** where the running service can be accessed.

WSDL:

A WSDL document can be thought of as a contract between a client and a server. It describes what the Web Service can do, where it can be found, and how to invoke it. Essentially, WSDL defines an XML grammar that describes Web Services (and in general, any network service) as collections of communications endpoints (that is, the client and the server) that are able to exchange messages with each other.
WSDL documents use the following elements:

- **definitions**. Associates the Web Service with its namespaces.
- **types.** A container for data type definitions, typically using a XML Schema Definition (XSD) or possibly some other type system.
- **message**. An abstract, typed definition of the data contained in the message.
- **operation**. An abstract description of an action that the Web Service supports.
- **portType.** The set of operations supported by one or more endpoints.
- **binding.** A specification of the protocol and data format for a particular portType.
- **port.** An endpoint, defined in terms of a binding and its network address (typically a URL). This is not a TCP/IP port, which is represented by a number.
- **service.** A collection of related endpoints.

The structure of a typical WSDL document is shown



WSDL Ports

The **<portType>** element is the most important WSDL element.

It describes a web service, the operations that can be performed, and the messages that are involved.

The <portType> element can be compared to a function library (or a module, or a class) in a traditional programming language.

WSDL Messages

The **<message>** element defines the data elements of an operation.

Each message can consist of one or more parts. The parts can be compared to the parameters of a function call in a traditional programming language.

WSDL Types

The **<types>** element defines the data types that are used by the web service.

For maximum platform neutrality, WSDL uses XML Schema syntax to define data types.

WSDL Bindings

The **<binding>** element defines the data format and protocol for each port type.

Calculator.xsd_1(http://localhost:8080/Calc/Calculator?xsd=1):

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:tns="http://p1/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
version="1.0"
targetNamespace="http://p1/">

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

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

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

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

<xs:complexType name="sub">
<xs:sequence>
<xs:element name="a" type="xs:int"/>
<xs:element name="b" type="xs:int"/>
</ xs:sequence>
</xs:complexType>

<xs:complexType name="subResponse">
<xs:sequence>
<xs:element name="return" type="xs:int"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="add">
<xs:sequence>
<xs:element name="a" type="xs:int"/>
<xs:element name="b" type="xs:int"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="addResponse">
<xs:sequence>
<xs:element name="return" type="xs:int"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

TESTING WEB SERVICES: (Web services->Calculator->Test Web Service)

add Method invocation

Method parameter(s)

Type	Value
Int	20
Int	10

Method returned

int : "**30**"

SOAP Request

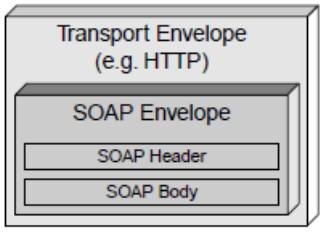
```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body>
      <ns2:add xmlns:ns2="http://p1/">
         <a>20</a>
         <b>10</b>
      </ns2:add>
    </S:Body>
</S:Envelope>
```

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body>
      <ns2:addResponse xmlns:ns2="http://p1/">
         <return>30</return>
      </ns2:addResponse>
   </S:Body>
</S:Envelope>
```

SOAP:

SOAP provides three key capabilities:
• SOAP is a messaging framework, consisting of an outer Envelope element that contains an optional Header element and a mandatory Body element.
• SOAP is an encoding format that describes how objects are encoded, serialized,and then decoded when received.
• SOAP is an RPC mechanism that enables objects to call methods of remote objects.



SOAP ENVELOPE ELEMENT:

The SOAP Envelope element is the mandatory top element of the XML document that represents the SOAP message being sent. It may contain namespace declarations as well as other attributes, which must be "namespace qualified." The Envelope element may also contain additional sub elements, which must also be namespace qualified and follow the Body element.

SOAP HEADER ELEMENT:

The SOAP Header element is optional and is used for extending messages without any sort of prior agreement between the two communicating parties. You might use the Header element for authentication, transaction support, payment information, or other capabilities that the SOAP specification doesn't provide.

Let's take a look at a typical Header element:

```
<SOAP-ENV:Header>
<t:Transaction xmlns:t="myURI"
SOAP-ENV:mustUnderstand="1">
3
</t:Transaction>
</SOAP-ENV:Header>
```

The SOAP Header element may also optionally contain the following attributes:

SOAP encodingStyle attribute:

A SOAP encodingStyle attribute, which would indicate the serialization rules for the header entries.

SOAP mustUnderstand attribute:

A SOAP mustUnderstand attribute (as in our example), which indicates whether it is optional or mandatory to process the header entry.

The value of the mustUnderstand attribute is either 1, indicating that the recipient must process the header entry, or 0, indicating that the header entry is optional. If this attribute doesn't appear, processing the header entry is assumed to be optional provide.

SOAP actor attribute:

A SOAP actor attribute, which indicates who is supposed to process the header entry and how they are supposed to process it.

SOAP BODY ELEMENT:

The mandatory Body element is an immediate child of the Envelope element and must immediately follow the Header element if a header is present. Each immediate child of the Body element is called a *body entry*.

SOAP FAULT ELEMENT:

The SOAP Fault element carries error messagesThe only Body entry explicitly defined in the SOAP specification is the Fault entry, used for reporting errors.

CREATING A JAVA WEB SERVICE CLIENT

a. Make a new project for the Client
   Create a new Web Application
   Give the name as CalcClient
b. Now Right Click on this client project
   Select a New Web Service Client
   In the Dialog box give the **WSDL url.**
   It will generate the client environment variables in the project
c. Now create a new servlet named hello.java
   Expand Web Service References -> call a Web Service Operation
d. Now select the Web Service name->Web Service port name and Web Service Operation Name
e. Click on Ok .
   It generates the code
   Now run the servlet .It will run the file call the web service and Print the result.

CODE:

Index.html:

```
<html>
   <head>
      <title> </title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form name="form1" action="Servlet1">
         Enter a: <input type="text" name="t1"><br>
          Enter b: <input type="text" name="t2"><br>
          <input type="submit" value="submit">

      </form>
   </body>
</html>
```

Servlet1.java:

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import p1.Calculator_Service;

@WebServlet(urlPatterns = {"/Serv1"})
public class Serv1 extends HttpServlet {
    @WebServiceRef(wsdlLocation = "WEB-INF/wsdl/localhost_8080/Calc/Calculator.wsdl")
    private Calculator_Service service;

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter())
    {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Serv1</title>");
            out.println("</head>");
            out.println("<body>");
            int x=Integer.parseInt(request.getParameter("t1"));
            int y=Integer.parseInt(request.getParameter("t2"));
            int sum = add(x, y);
            int diff = sub(x, y);
```

```
            out.println("<h1>Sum is:" + sum + "</h1>");
            out.println("<h1>Diff is:" + diff + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
private int add(int a, int b)
    {
        p1.Calculator port = service.getCalculatorPort();
        return port.add(a, b);
    }
 private int sub(int a, int b)
    {
        p1.Calculator port = service.getCalculatorPort();
        return port.sub(a, b);
    }
}
```