

DEVELOPING WEB SERVICES WITH JAVA

DESIGN WEB SERVICE CLIENT

CONTENTS

- Web Service Clients
- Modes of Communications
- Locating and Accessing Web Services
- Exception Handling Mechanism
- Steps to create Web Services with J2EE 1.4 Framework using NetBeans
- Workshops
- Exercises

DYNAMIC PROXY

- Creates a proxy of the service interface and then uses this interface to access the Service's methods.
- Support the client do not create a stub.
- Accesses Web Services using their interface proxy enabling client applications to be portable.
- Steps to create client
 - Create a *Service* object.
 - Import the interface class created by the *wscompile* tool.
 - Create a *serviceFactory* object.
 - Then, using the *serviceFactory* object's *createService()* method the developer needs to create a *Service* object that requires the WSDL file location, and a QName constructor instance. QName in turn requires a namespace URI and the service name.
 - Create a proxy using the Service objects *getPort()* method.
 - Once the Service object is created, using getPort() method to create the proxy that can be used to access the Web Service's methods once it has been typecast with the imported interface class.
 - Type cast the proxy as an interface.
 - Invoking Web Service using proxy object.

```
Service helloService =  
serviceFactory.createService(helloWsdUrl,  
    new QName(namespaceUri, serviceName));
```

DYNAMIC PROXY

```
package dynamicproxy;

import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import dynamicproxy.MyDynamicGenClient.MyHelloServiceRPC;

public class DynamicProxyHello {

    public static void main(String[] args) {
        try {

            String urlString =
                "http://localhost:80/MyHelloService/MyHelloService?WSDL";
            String namespaceUri = "urn:MyHelloService/wsdl";
            String serviceName = "MyHelloService";
            String portName = "MyHelloServiceRPCPort";

            URL helloWsdUrl = new URL(urlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service helloService =
                serviceFactory.createService(helloWsdUrl,
                    new QName(namespaceUri, serviceName));

            MyHelloServiceRPC myProxy =
                (MyHelloServiceRPC) helloService.getPort(
                    new QName(namespaceUri, portName),
                    MyHelloServiceRPC.class);

            System.out.println(myProxy.sayHello("Buzz"));

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

DYNAMIC INVOCATION INTERFACE

- Uses *Call* objects to dynamically invoke a Web Service. Doing so enables the client to invoke methods of Service even without knowing their addresses until runtime.
- Provides the client developer with complete control over the client application.
- Is the only method that supports one-way invocation (except return values and out parameters are possible).
- *Disadvantages*: The client developer needs to develop very complicated code.
- Steps to create DII
 - Create a *Service* object.
 - Create a *QName* class instance using the generates interface class.
 - The service can be created using the *createService()* method with a QName object (service name, this name is specified in WSDL file) as the input parameter.
 - Create the *Call* object and set its address.
 - The *Call* object will enable the client to access to the Web Service methods.
 - Set the Call object properties.
 - Set its *TARGET_ENDPOINT_ADDRESS* and a few other properties such as *SOAPACTION_USE_PROPERTY*, *SOAPACTION_URI_PROPERTY* and *ENCODING_STYLE_PROPERTY*.
 - Set Web Service operation name.
 - Invoke Web Service method.

DYNAMIC INVOCATION INTERFACE

```
package dii;

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

public class DIHello {

    private static String qnameService = "MyHelloService";
    private static String qnamePort = "MyHelloServiceRPC";
    private static String endpoint =
        "http://localhost:80/MyHelloService/MyHelloService";
```

```
    private static String BODY_NAMESPACE_VALUE =
        "urn:MyHelloService/wsdl";
    private static String ENCODING_STYLE_PROPERTY =
        "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD =
        "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING =
        "http://schemas.xmlsoap.org/soap/encoding/";
```

```
    public static void main(String[] args) {
        try {
```

```
            ServiceFactory factory =
                ServiceFactory.newInstance();
            Service service =
                factory.createService(new QName(qnameService));
```

```
            QName port = new QName(qnamePort);
```

```
            Call call = service.createCall(port);
            call.setTargetEndpointAddress(endpoint);
```

The name of this interface is designated by the portType element:
<portType name="MyHelloServiceRPC">

Sets these properties on the Call object

```
call.setProperty(Call.SOAPACTION_USE_PROPERTY,
    new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(ENCODING_STYLE_PROPERTY,
    URI_ENCODING);
QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);
```

Specifies the method's return type, name, and parameter

```
call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
    "sayHello"));
call.addParameter("String_1", QNAME_TYPE_STRING,
    ParameterMode.IN);
String[] params = { "Murphy" };
```

```
String result = (String)call.invoke(params);
System.out.println(result);
```

```
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

In the WSDL file, this address is specified by the <soap:address> element:

```
<service name="MyHelloService">
    <port name="MyHelloServiceRPCPort" binding="tns:MyHelloServiceRPCBinding">
        <soap:address location="http://localhost:80/MyHelloService/MyHelloService"/>
    </port>
</service>
```

STATIC STUB

- Involves creating stub classes that enable the client application and the Web Service to communicate.
- Is then typecase as an interface, which in turn is used to access the Service's method.
- Can directly code their clients against the stub also makes the Static Stub method the easiest to code and implement.
- Allows creation, insertion, modification, and deletion of a range of content in elements like document, document fragment, or attribute.
- **Disadvantages:** Any change in the service interface would require the client developer to start all over again.
- Steps to create Static Stub.
 - Retrieve the service endpoint interface implementation in a Stub object.
 - Import the `javax.xml.rpc.Stub` interface in order to implement the stub.
 - Retrieve an instance of the service implementation class (`ServiceName_impl()`) generated by *wscompile*.
 - Typecast the stub as an interface.
 - Set the endpoint address for the stub object.
 - The `ServiceName_impl()` class's needs to set the `ENDPOINT_ADDRESS_PROPERTY` with the service's endpoint address.
 - After setting the endpoint address, the developer needs to create an instance of the service interface class using the Stub object.
 - Access Web Service methods using stub object.

STATIC STUB

```
package staticstub;

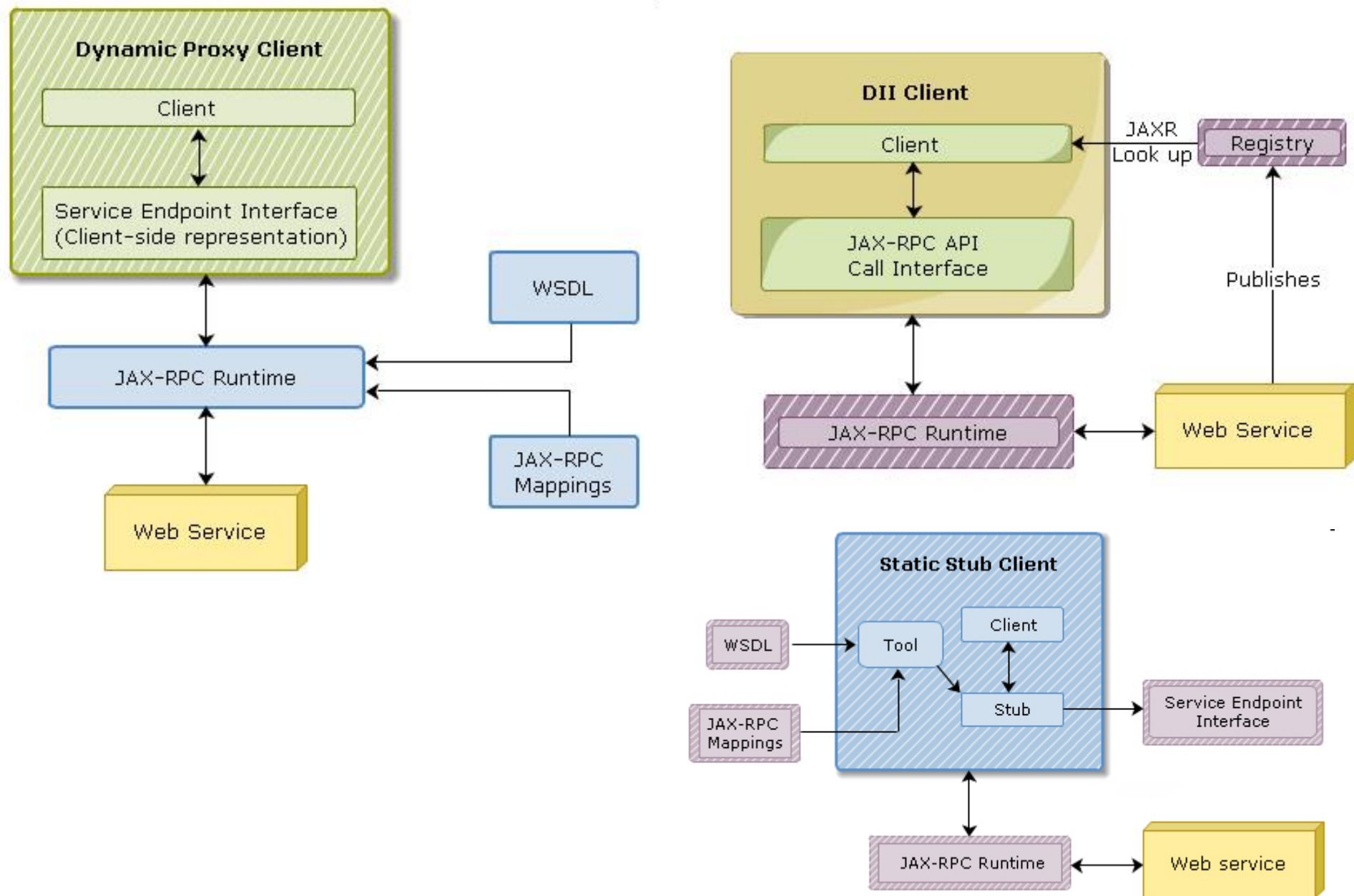
import javax.xml.rpc.Stub;
import staticstub.MyStaticGenClient.MyHelloService_Impl;
import staticstub.MyStaticGenClient.MyHelloServiceRPC;

public class StaticStubHello {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            MyHelloServiceRPC hello = (MyHelloServiceRPC) stub;
            System.out.println(hello.sayHello("Duke"));
            System.out.println(hello.sayGoodbye("Jake"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

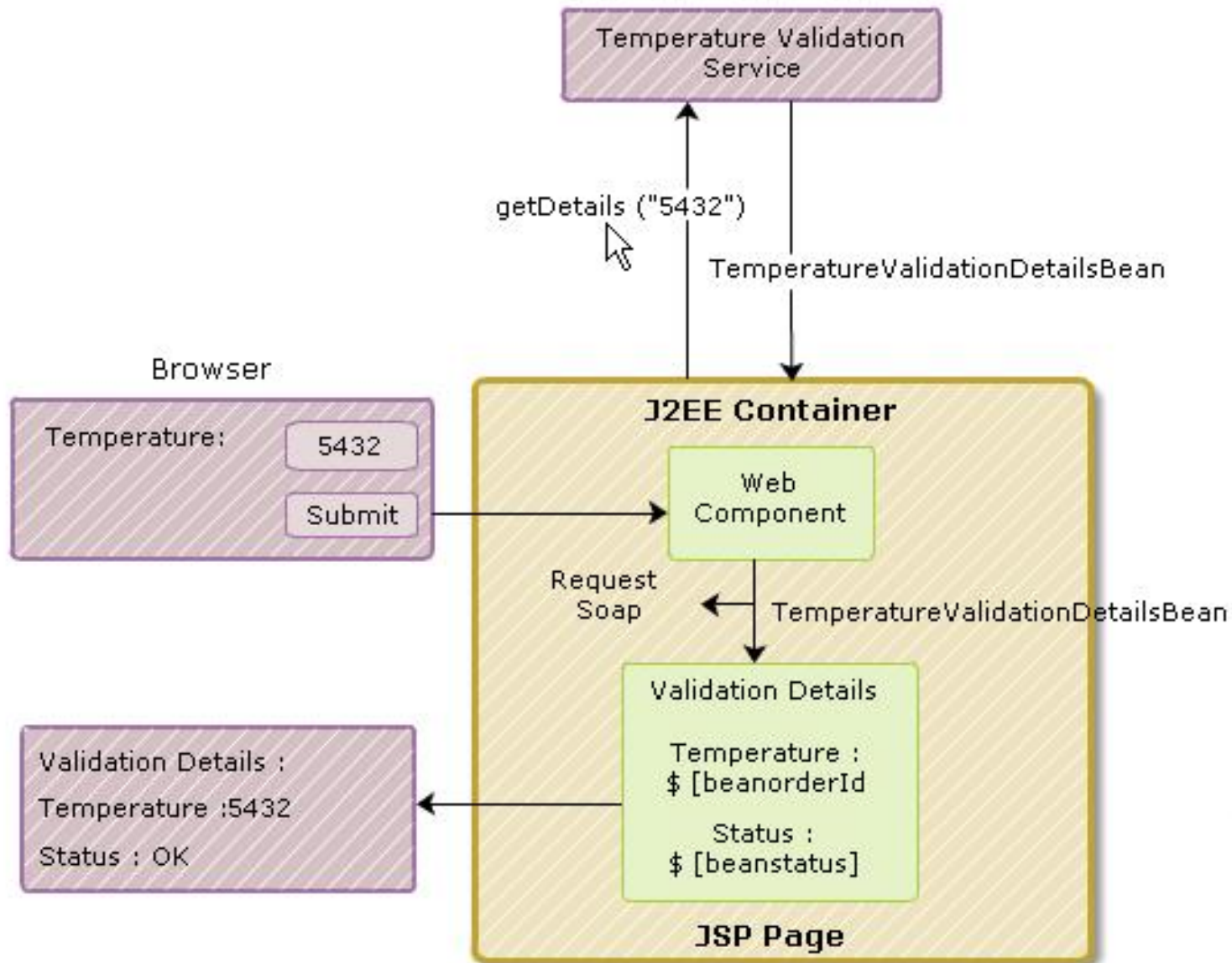
    private static Stub createProxy() {
        // Note: MyHelloService_Impl is implementation-specific.
        return (Stub)
            (new MyHelloService_Impl().getMyHelloServiceRPCPort());
    }
}
```

Casts stub to the service definition interface, MyHelloServiceRPC

WEB SERVICE CLIENT



J2EE APPLICATION CLIENT



MODES OF COMMUNICATIONS

- **Dynamic Proxy Communication.**
 - The client directly accesses the Web Service through its proxy, that is, the client-side representation of the service endpoint interface.
 - The Web Service client is programmed to interact with the endpoint service. This ensures that the client application is portable across different JAX-RPC runtimes. The JAX-RPC runtime uses WSDL document and the JAX-RPC mapping files.
 - This invocation mode should be used only when the WSDL contains primitive schema types.
 - Developers using dynamic proxies must create Java classes from client-side interface matching with service endpoint interface
- **DII Communication.**
 - The DII approach enables clients to dynamically access Web Services through programmatic invocation of JAX-RPC requests.
 - Based on the Web Service WSDL document, a tool is used to create the service endpoint interface, the stub and other necessary class files. Even though the need for the JAX-RPC APIs is ruled out by the extra class files, DII uses them.
 - The DII client uses JAXR to search the registry for services.
 - The client constructs a call from the information it receives from the service registry.
 - The client then uses the constructed call to access the Web Service. Thus, like Dynamic Proxies, DII also uses JAX-RPC for its communication with the service endpoint interface

MODES OF COMMUNICATIONS (cont)

- Static Stub Communication.
 - The JAX-RPC runtime tool generates the stub. The tool uses the WSDL document and the JAX-RPC mapping files to generate the stub.
 - The stub can be considered as a local object that acts as a proxy for the service endpoint. It enables communication between the client and the service. It converts client requests to SOAP messages and passes them onto the service, it reconverts the SOAP messages it receives from the service and passes them back as responses to the client.

LOCATING & ACCESSING

- Dynamic Proxy.
 - Uses Java Naming and Directory Interface's (JNDI) `InitialContext.lookup()` method to locate the Web Service.
 - Does not cast the JNDI reference as a service implementation class. Instead, the JNDI reference is cast as a *javax.xml.rpc.Service* interface.
 - Casting the JNDI reference as a service interface removes the dependency on the stub generated in the Static Stub approach. This makes the client portable. The client developer uses the *getport()* method to retrieve the service's port address. Once the service port number is obtained, the developer can directly call the service's method as and when required.
- DII.
 - Enables Web Service Clients to dynamically discover and utilize Web Service during runtime. The *javax.xml.rpc.Call* interface enables clients to dynamically invoke JAX-RPC services.
 - The DII approach uses a Call object to access a Web Service's methods.
 - The approach requires the Call object to be programmed in such a manner that the client application would only require the Web Service's details at runtime.
 - Once the Web Service's details are retrieved the Call object's *setTargetEndpointAddress()*, *setProperty()*, *setReturnType()*, *setOperationName()*, and *addParameter()* methods are used to equip the Call object to be used by the client.
 - Once these properties are set the client can use the Call object's *invoke()* method to access the Web Service's methods

LOCATING & ACCESSING (cont)

- Static Stub.
 - Approach uses JNDI, `InitialContext.lookup()` method to locate the Web Service.
 - The client creates a Service object by casting the JNDI reference as a service. After creating the Service object, the client uses the service's `getPort()` method to retrieve an instance of the generated stub implementation.
 - Using the JNDI reference to create a stub implementation makes causes the client implementation-specific. This dependence does not allow the client application to be portable across various systems. This is the biggest drawback introduced by the Static Stub approach.

COMPARE

Static Stub	Dynamic Proxy	Dynamic Invoke Invocation
<p>Simplicity. Only two lines of code are required to access and invoke a WS's operation.</p>	<p>Portable, vendor-independent code.</p>	<p>Client can call a remote procedure without development-time knowledge of the WSDL URI => the code easy to modify if the WS details change. Runtime classes generated by WSDL to Java mapping tools are not required.</p>
<p>Need to know the WSDL URL at development-time and run your WSDL to Java mapping tool. These stubs are not portable because they depend on implementation classes. The design of portable stubs is out-of-scope for JAX-RPC 1.0 and 1.1.</p>	<p>Need to know the WSDL URL at development-time and need to run your WSDL to Java mapping tool against the WSDL document before runtime. If the WSDL URL is likely to change, you should use the DII method instead.</p>	<p>Configuring the Call instance is complex.</p>

EXCEPTION HANDLING MECHANISM

- There are two types of exceptions:
 - *System*.
 - A *javax.xml.rpc.ServiceException* is thrown by dynamic proxy clients. (Insufficient data to create the proxy by getPort method)
 - *RemoteException* & *javax.xml.rpc.JAXRException* are thrown by DII Call interfaces. (Unavailability of Service, network failures)
 - Static Stub clients mostly face problems during the configuration of the stub. Configuration errors like invalid property names, invalid property values, type mismatch, result in *javax.xml.rpc.JAXRException*.
 - *Service-specific*.
 - Are thrown by faults or errors generated by the client application itself.
 - These exceptions are also called checked exceptions in client applications. These faults or errors are a result of improper data passed to the Web Service. Since they are generated by the application itself, it is easy to determine these errors, and also to handle them. These exceptions are generally listed as operation elements in a WSDL file, and these are known as wsdl:fault elements.
 - JAX-RPC tools can be used to map faults or errors to Java objects. These tools generate necessary exception classes and parameters to handle. These exception classes extend java.lang.Exception. The client application is responsible for handling these checked exceptions. The client application should also provide means to recover from such exceptions. DII communication mode returns all exceptions as java.rmi.RemoteException.

EXCEPTION HANDLING MECHANISM

- In case of J2EE Web Components, clients may handle service exceptions as unchecked applications, such as `javax.servlet.ServletException` or may divert it to an error page.
- Client developers can resort to boundary checking for input values, for example, they can check if credit card numbers that are entered are 16-digit integers or particular range. Developers can use JavaScript to validate the boundaries before sending requests to a service. This validation will minimize multiple trips to the service, thereby reducing network traffic, and increasing service access speed.

JAX-RPC SERVICE ENDPOINT

- Features:
 - Defines a common programming model for both Web Service clients and endpoints in J2EE.
 - Can be used to access Web Services that run in non-Java environments.
 - Can be used to host Web Service endpoints so that non-Java client applications too can access them.
 - Is designed as a Java API so that J2EE applications could interoperate
- Implementation
 - Defining the Remote Interface
 - A remote interface is defined with containing all the methods that would be exposed as a part of the service.
 - The interface must extend the `java.rmi.Remote` interface and its methods should throw `java.rmi.RemoteException`.
 - Implementing the Remote Interface
 - The class implementing the interface must be defined.
 - The class must contain the actual implementation of all the methods declared in the remote interface.
 - Building the Service
 - Compiling the .java files of remote interface and implementation class.
 - Next, the `wscompile` tool is used to create the WSDL file the mapping file .WSDL file describes the Web Service.
 - Packaging and Deployment: the class files, WSDL file and mapping file are packaged into a WAR file and deployed on a server such as Sun's Application Server

STEPS TO CREATE WS WITH J2EE 1.4

- **Step 1:** Creating Web Application
- **Step 2:** Adding Web Services to Web Application
- **Step 3:** Adding Methods/ Operations to Web Services
- **Step 4:** Configuring the Web Services to access
- **Step 5:** Build the Web Services, then deploy the it to server
- **Step 6:** Testing Web Services, generate the WSDL

STEPS TO CREATE WS WITH J2EE 1.4

- **Step 1: Creating Web Application**

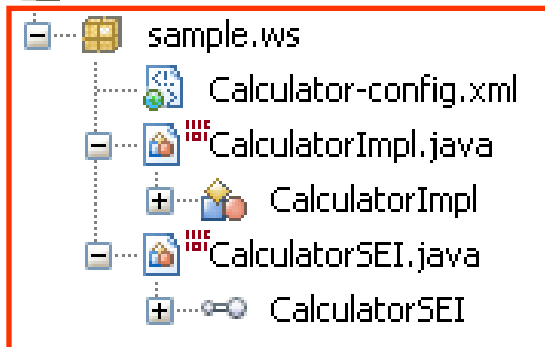
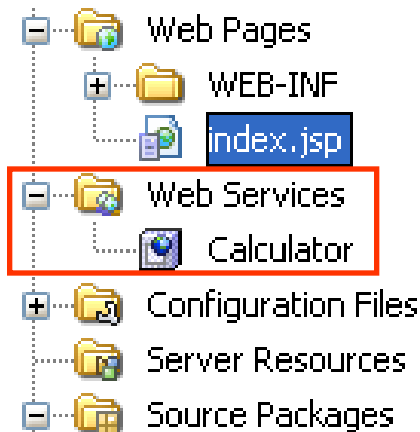
The screenshot shows the 'New Web Application' dialog box in the Eclipse IDE. The 'Steps' pane on the left indicates the current step is '2. Name and Location'. The 'Name and Location' tab is active, showing the following fields and options:

- Project Name:** Calculate1_4WS (highlighted with a red box and an arrow pointing to the text 'Fill your Web App Name').
- Project Location:** G:\Laptrinh\Servlet (with a 'Browse...' button).
- Project Folder:** G:\Laptrinh\Servlet\Calculate1_4WS.
- Source Structure:** Java BluePrints (dropdown menu).
- Add to Enterprise Application:** <None> (dropdown menu).
- Server:** Sun Java System Application Server 9 (dropdown menu, highlighted with a red box and an arrow pointing to the text 'Choose the Sun Server instead of Tomcat').
- Java EE Version:** J2EE 1.4 (dropdown menu, highlighted with a red box and an arrow pointing to the text 'Choose the J2EE 1.4').
- Context Path:** /Calculate1_4WS (highlighted with a red box and an arrow pointing to the text 'Rename the context path if necessary. Should not be changed').
- Recommendation:** Source Level 1.4 should be used in J2EE 1.4 projects.
- Set Source Level to 1.4:** ☒ (checked).
- Set as Main Project:** ☒ (checked).

At the bottom of the dialog, the 'Finish' button is highlighted with a red box and an arrow pointing to the text 'Click Finish'. Other buttons include '< Back', 'Next >', 'Cancel', and 'Help'.

STEPS TO CREATE WS WITH J2EE 1.4

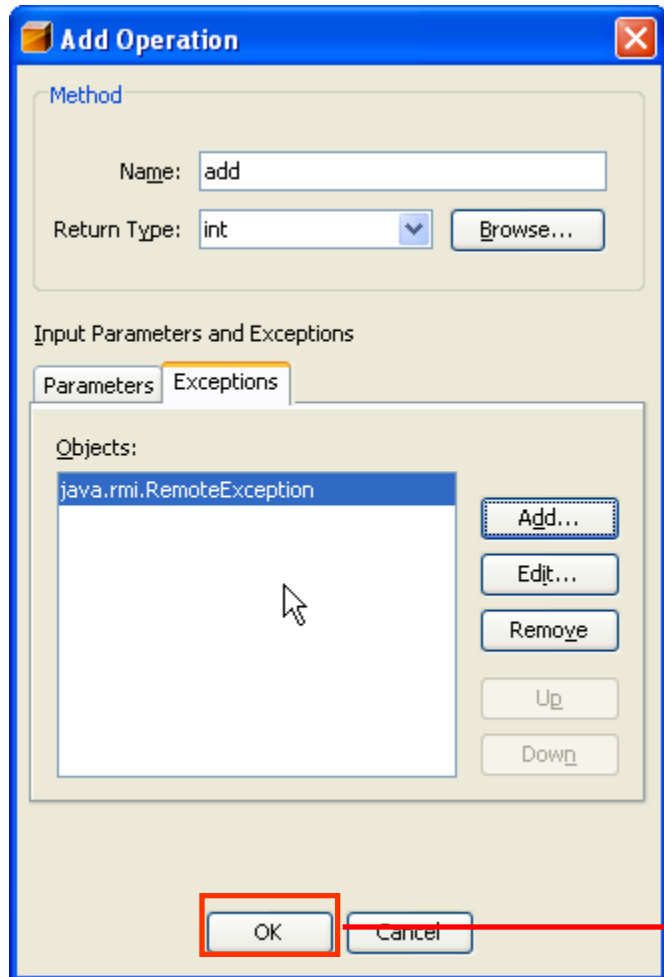
- **Step 2:** Adding Web Services to Web Application
 - Same as the first lesson



```
public class CalculatorImpl implements CalculatorSEI {  
  
    // Enter web service operations here. (Popup menu: Web Service->Add Operation)  
}  
  
public interface CalculatorSEI extends java.rmi.Remote {  
  
}
```

STEPS TO CREATE WS WITH J2EE 1.4

- **Step 3:** Adding Methods/ Operations to Web Services
 - Same as the first lesson, but adding the Exception as



Click Ok Button

STEPS TO CREATE WS WITH J2EE 1.4

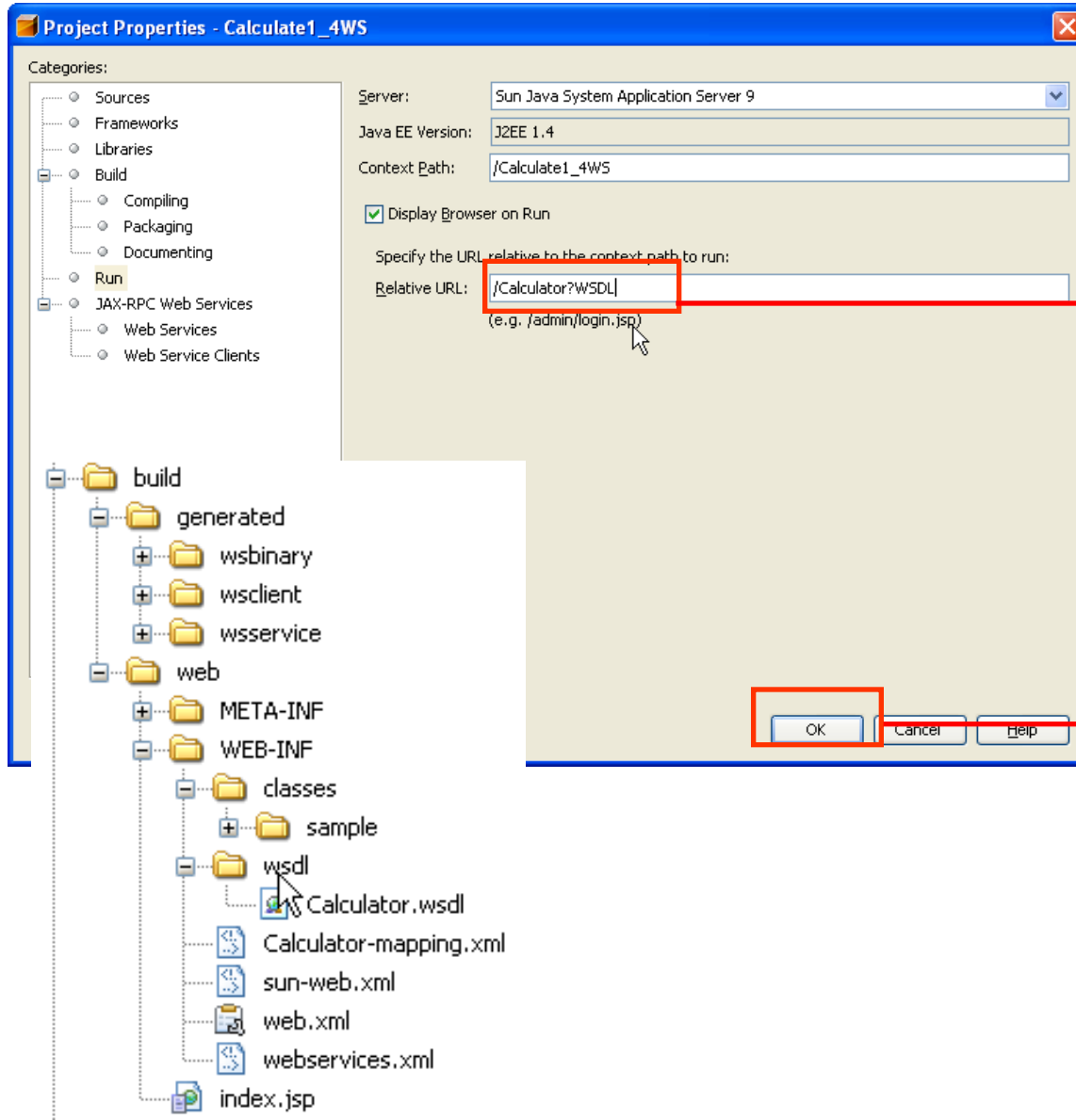
- **Step 3: Adding Methods/ Operations to Web Services(cont)**
 - Same as the first lesson, then result is automatically generated as

```
13 public interface CalculatorSEI extends Remote {
14     /**
15      * Web service operation
16      */
17     public int add (int num1, int num2) throws RemoteException;
18
19     /**
20      * Web service operation
21      */
22     public int subtract (int num1, int num2) throws java.rmi.RemoteException;
23 }
24 }
```

```
11 public class CalculatorImpl implements CalculatorSEI {
12     /**
13      * Web service operation
14      */
15     public int add (int num1, int num2) throws RemoteException {
16         // TODO implement operation
17         return num1 + num2;
18     }
19
20     /**
21      * Web service operation
22      */
23     public int subtract (int num1, int num2) throws java.rmi.RemoteException {
24         // TODO implement operation
25         return num1 - num2;
26     }
27 }
28 }
```

STEPS TO CREATE WS WITH J2EE 1.4

- **Step 4:** Configuring the Web Services to access

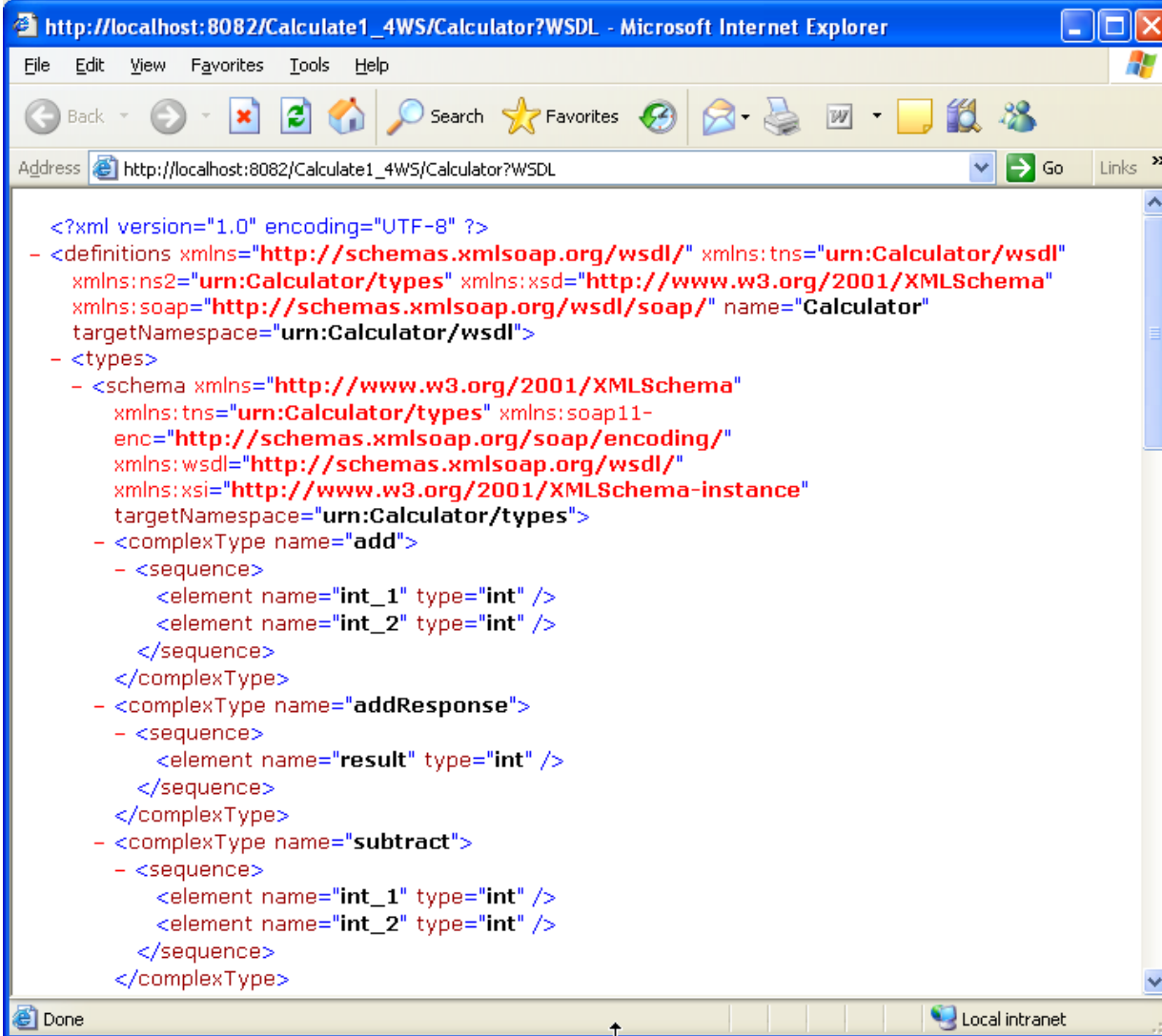


Set default the service name?WSDL for testing of WS

Click Ok

STEPS TO CREATE WS WITH J2EE 1.4

- Step 5: Building and Deploying WS
- Step 6: Running and Testing WS



The screenshot shows a Microsoft Internet Explorer window with the address bar displaying `http://localhost:8082/Calculate1_4WS/Calculator?WSDL`. The main content area displays the XML WSDL for a web service named "Calculator". The XML is color-coded, with namespaces in red and other elements in blue. The WSDL defines two complex types: "add" and "subtract", each containing two integer elements. It also defines an "addResponse" complex type containing a "result" integer element.

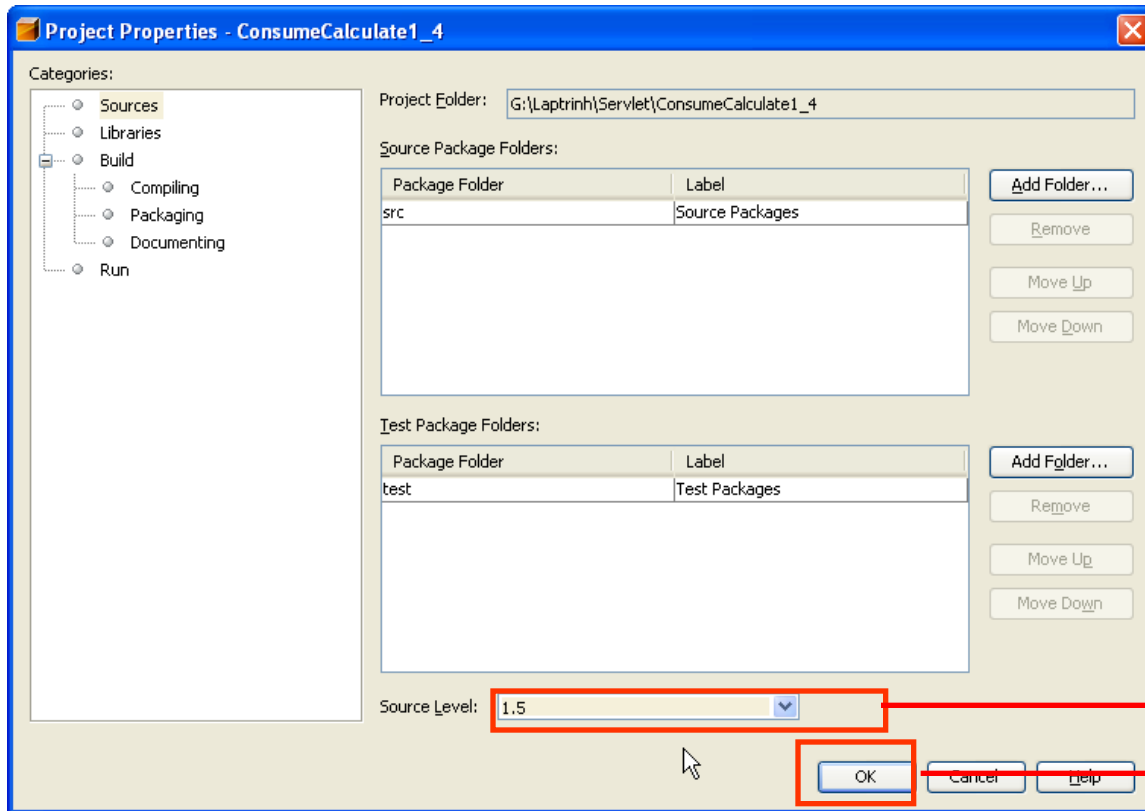
```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:Calculator/wsdl"
  xmlns:ns2="urn:Calculator/types" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" name="Calculator"
  targetNamespace="urn:Calculator/wsdl">
- <types>
  - <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn:Calculator/types" xmlns:soap11-
      enc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      targetNamespace="urn:Calculator/types">
    - <complexType name="add">
      - <sequence>
        <element name="int_1" type="int" />
        <element name="int_2" type="int" />
      </sequence>
    </complexType>
    - <complexType name="addResponse">
      - <sequence>
        <element name="result" type="int" />
      </sequence>
    </complexType>
    - <complexType name="subtract">
      - <sequence>
        <element name="int_1" type="int" />
        <element name="int_2" type="int" />
      </sequence>
    </complexType>
```

STEPS TO CREATE WS CLIENT USING DYNAMIC PROXY

- **Step 1:** Creating Web/Swing Application
- **Step 2:** Adding Web Services Client to Application
- **Step 3:** Write code to access the WS
- **Step 4:** Configuring the Application
- **Step 5:** Running the application

DYNAMIC PROXY

- **Step 1: Creating Web/Swing Application**
 - After the creating the application, the source of the Application should be changed to 1.5 by configuring the Properties of Project



Set Level source to 1.5

Click Ok

DYNAMIC PROXY

- **Step 2:** Adding Web Services Client to Application

New Web Service Client

Steps

1. Choose File Type
2. **WSDL and Client Location**

WSDL and Client Location

Specify the WSDL file of the Web Service.

☐ Project:

☐ Local File:

☒ WSDL URL:

Specify a location for the client.

Project:

Package:

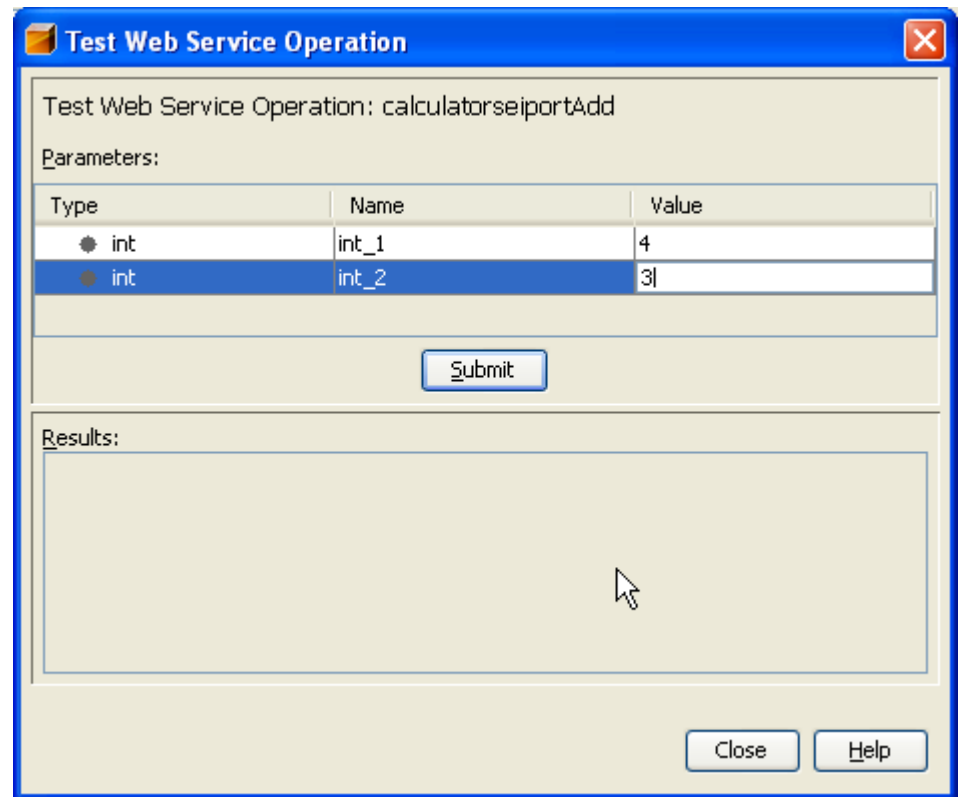
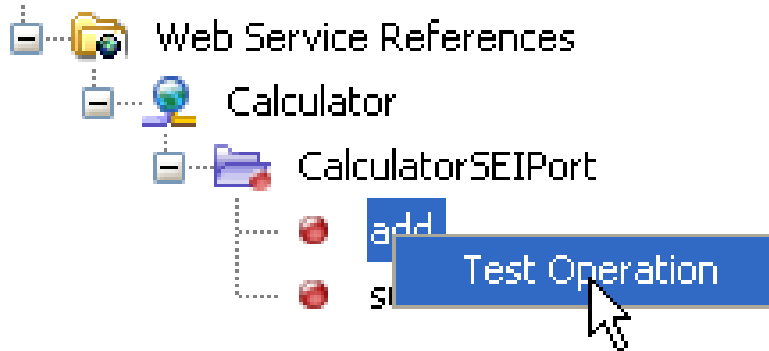
JAX Version:

Web Service References

- Calculator
 - CalculatorSEIPort
 - add
 - subtract

DYNAMIC PROXY


- **Step 2:** Adding Web Services Client to Application (cont)
 - Test Operation on Client by right click on service, choose Test Operation



DYNAMIC PROXY

- **Step 3: Writing Code**

```
public class DynaProxy {  
    private static String url =  
        "http://localhost:8082/Calculate1_4WS/Calculator?WSDL";  
    private static String uri = "urn:Calculator/wsdl";  
    private static String serviceName = "Calculator";  
    private static String portName = "CalculatorSEIPort";  
}
```

Address  http://localhost:8082/Calculate1_4WS/Calculator?WSDL

```
<?xml version="1.0" encoding="UTF-8" ?>  
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:Calculator/wsdl" xmlns:ns2="urn:Calculator/types"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" name="Calculator"  
  targetNamespace="urn:Calculator/wsdl">  
- <service name="Calculator">  
  - <port name="CalculatorSEIPort" binding="tns:CalculatorSEIBinding">  
    <soap:address location="http://192.168.214.129:8082/Calculate1_4WS/Calculator"  
      xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" />  
  </port>  
</service>
```

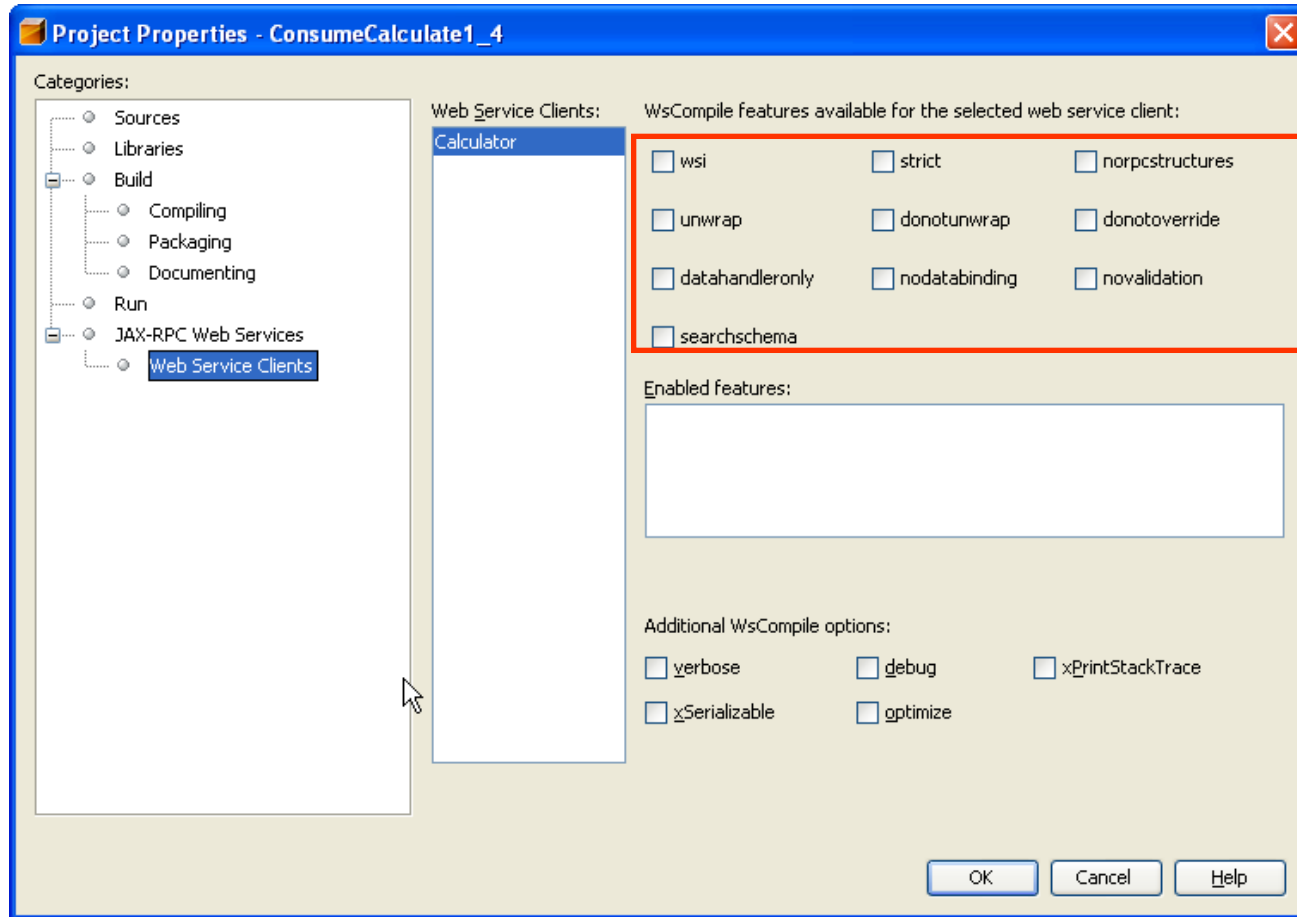
DYNAMIC PROXY

- **Step 3: Writing Code (cont)**

```
public static void main (String[] args) {
    int choice = 0, num1 = 0, num2 = 0;
    try{URL converterWSDLURL = new URL(url);
        ServiceFactory serviceFactory = ServiceFactory.newInstance ();
        QName serviceName = new QName(uri, serviceName);
        Service calculatorService = serviceFactory.createService (converterWSDLURL, serviceName);
        QName name = new QName(uri, portName);
        Object objPort = calculatorService.getPort (name, CalculatorSEI.class);
        CalculatorSEI myProxy = (CalculatorSEI)objPort;
        Scanner input = new Scanner(System.in);
        do { System.out.println("Calculator Program");
            System.out.println("Select one operation");
            System.out.println("1. Add 2 Number");
            System.out.println("2. Subtract 2 Number");
            System.out.println("3. Exit");
            choice = input.nextInt ();
            switch(choice){
                case 1: System.out.println("Enter num1 "); num1 = input.nextInt ();
                    System.out.println("Enter num2"); num2 = input.nextInt ();
                    System.out.println(num1 + " + " + num2 + " = " + myProxy.add (num1, num2));
                    break;
                case 2: same as case 1
                case 3: break;
                default: System.out.println("\nInvalid choice"); break;
            }
        }while(choice!=3);
    }catch(Exception e){ e.printStackTrace ();}
}
```

DYNAMIC PROXY

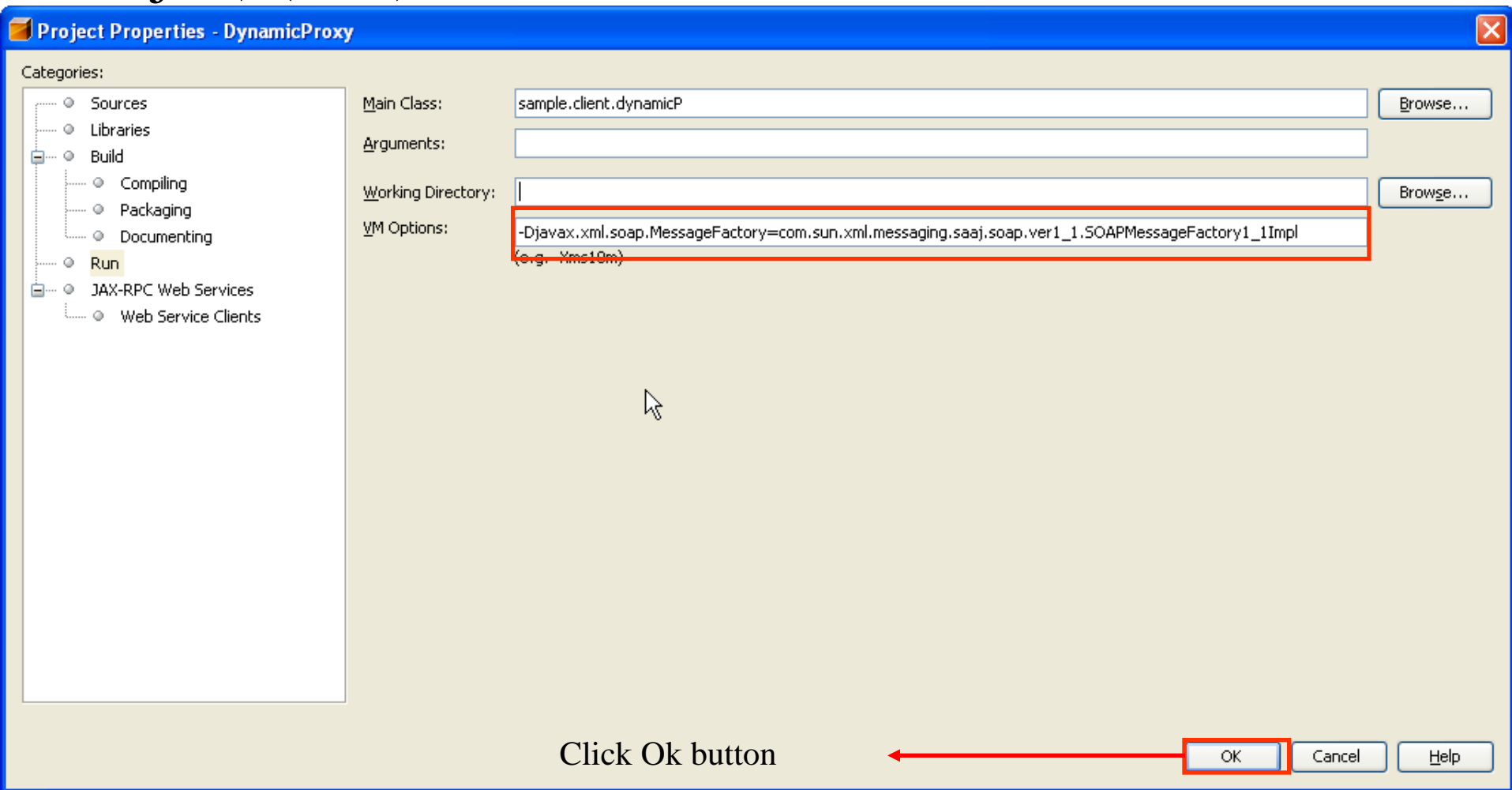
- **Step 4:** Configuring the Application (using Properties of Project)



Uncheck all of check box

DYNAMIC PROXY

- **Step 4:** Configuring the Application (using Properties of Project) (cont)



DYNAMIC PROXY

- **Step 5: Building and Running Project**

```
run:
Calculator Program
Select one operation
1. Add 2 Number
2. Subtract 2 Number
3. Exit |
```

Input

```
1
Enter num1
3
Enter num2
4
3 + 4 = 7
Calculator Program
Select one operation
1. Add 2 Number
2. Subtract 2 Number
3. Exit |
```

Input

4

```
2
Enter num1
6
Enter num2
5
6 - 5 = 1
Calculator Program
Select one operation
1. Add 2 Number
2. Subtract 2 Number
3. Exit |
```

Input

5

DYNAMIC INVOCATION INVOKE

- **The steps is same as the Dynamic**

DYNAMIC INVOCATION INVOKE

```
public class DynaII {
    private static String qnameService = "Calculator";
    private static String qnamePort = "CalculatorSEIPort";
    private static String uri = "urn:Calculator/wsdl";
    private static String ENCODING_STYLE_PROPERTY =
        "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";
    public static void main (String[] args) {
        int choice =0;
        int num1 = 0;
        int num2 = 0;
        try {
            ServiceFactory factory = ServiceFactory.newInstance ();
            Service service = factory.createService (new QName(qnameService));
            QName port = new QName(qnamePort);
            Call call = service.createCall (port);
            call.setTargetEndpointAddress ("http://localhost:8082/Calculate1_4WS/Calculator");
            call.setProperty (Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
            call.setProperty (Call.SOAPACTION_URI_PROPERTY, "");
            call.setProperty (ENCODING_STYLE_PROPERTY, URI_ENCODING);
            call.setProperty ("javax.xml.rpc.soap.operation.style", "rpc");
```

DYNAMIC INVOCATION INVOKE

```
QName QNAME_TYPE_INT = new QName(NS_XSD, "int");
```

```
Scanner input = new Scanner(System.in);
```

```
do{ Integer[] params = {new Integer(0), new Integer(0)};
```

```
    saem as the dynamic proxy
```

```
    choice = input.nextInt ();
```

```
    switch (choice){
```

```
        case 1: same as the dynamic proxy
```

```
            params[0] = new Integer(num1);
```

```
            params[1] = new Integer(num2);
```

```
            call.setReturnType (QNAME_TYPE_INT);
```

```
            call.setOperationName (new QName(uri, "add"));
```

```
            call.addParameter ("int_1", QNAME_TYPE_INT, ParameterMode.IN);
```

```
            call.addParameter ("int_2", QNAME_TYPE_INT, ParameterMode.IN);
```

```
            System.out.println(num1 + " + " + num2 + " = ");
```

```
            System.out.println(((Integer)call.invoke (params)).intValue () + "\n");
```

```
            call.removeAllParameters ();
```

```
            break;
```

```
        case 2: same as case 1
```

```
        case 3: break;
```

```
        default: System.out.println("\nInvalid choice"); break;
```

```
    }
```

```
    params = null;
```

```
    }while(choice!=3);
```

```
    }catch(RemoteException e){ e.printStackTrace ();
```

```
    }catch(ServiceException e){e.printStackTrace ();
```

```
    }catch(Exception e){e.printStackTrace ();}
```

```
}}
```

WORKSHOP ACTIVITIES

Designing Web Service Clients			X
Creating Static Stub Client	 Show Me	 Let Me Try	
Invoking Web Service Operations Using Static Stub Instance	 Show Me	 Let Me Try	
Building and Running the Application	 Show Me	 Let Me Try	
Creating Dynamic Proxy Client	 Show Me	 Let Me Try	
Invoking Web Service Operations Using Dynamic Proxy Instance	 Show Me	 Let Me Try	
Building and Running the Application	 Show Me	 Let Me Try	
Creating Dynamic Invocation Interface Client	 Show Me	 Let Me Try	
Invoking Web Service Operations using Dynamic Invocation Interface Instance	 Show Me	 Let Me Try	
Building and Running the Application	 Show Me	 Let Me Try	

Building the WebService Client consume WS that can convert from C to F degree and interaction using J2EE 1.4 on Sun Java Application Server System 9

- Static Stub
- Dynamic Proxy & DII

EXERCISES

- Write the Web Service Client with all the exercises in first lesson