Lab Exercise

Transformers - Lab 4

No, Spring does not have Autobots, but the concept – turn something from one thing to another thing - is provided for in Spring Integration. Spring Integration (SI) transformers turn one type of message into another. In the world of integration, data providers and data consumers don't always speak the same language. So transformers provide SI applications the means to convert messages between formats to facilitate non-homogeneous message exchange. For example, a producer may provide information in XML format. It is the data your application needs, but it would like that data in JSON form. A SI transformer can perform that message conversion.

In this lab, you explore several types of SI transformers – some provided by SI out of the box. As you have learned with other SI components, you can also create your own custom transformer.

Specifically, in this lab you will:

- Configure and use a message payload transformer.
- Define a custom transformer.
- Explore the use of annotations to reduce SI component XML configuration.
- Examine the use of an XML to object transformer – otherwise known as an unmarshalling transformer.

Lab solution folder: ExpressSpringIntegration\lab4\lab4-transformer-solution & lab4-xml-transformer-solution

Scenario – Transform String Messages

In the first part of this lab, you some simple string message to string message transformers. That is, these transformers take a message with string payload from a channel (called the source message in SI), change the string payload of a message, and put a message with the altered string into another channel (called the target message in SI). These simple examples will help you understand the basic configuration of a transformer and allow you to see how to create your own custom transformer.

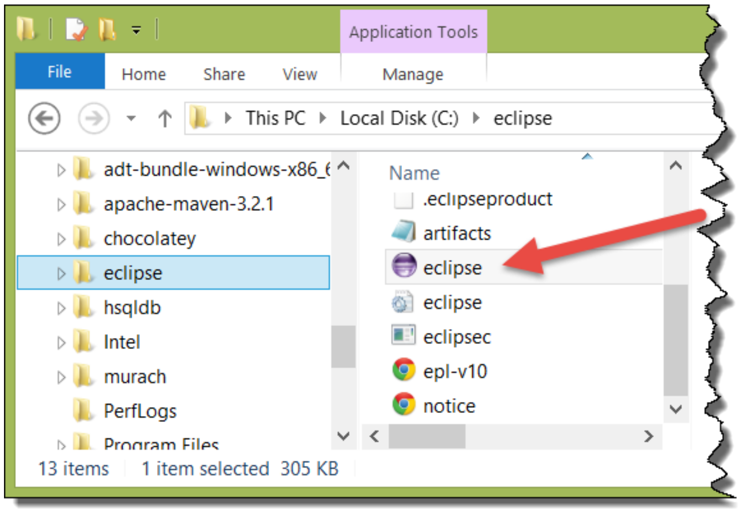
Step 1: Import the Maven Project

A Maven Project containing the base code for a string transformer application has already been created for you. You will use this project to begin your exploration of transformers.

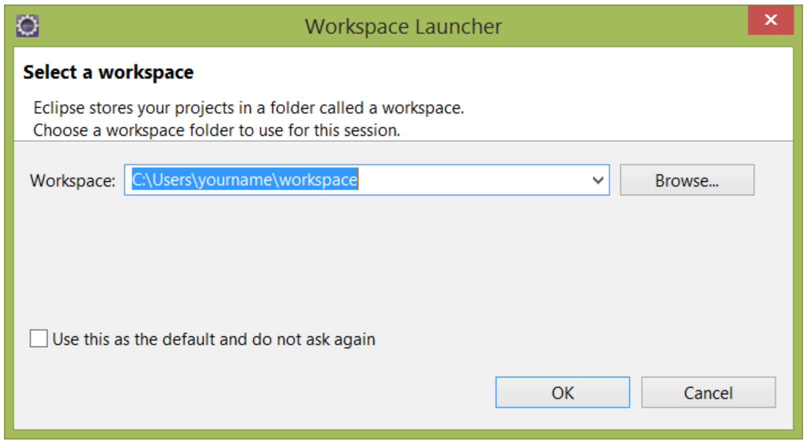
1.1 Start the Eclipse-based IDE. Locate and start Eclipse (or Eclipse-based) IDE.

1.1.1 Locate the Eclipse folder.

1.1.2 Start Eclipse by double clicking on the eclipse.exe icon (as highlighted in the image below).



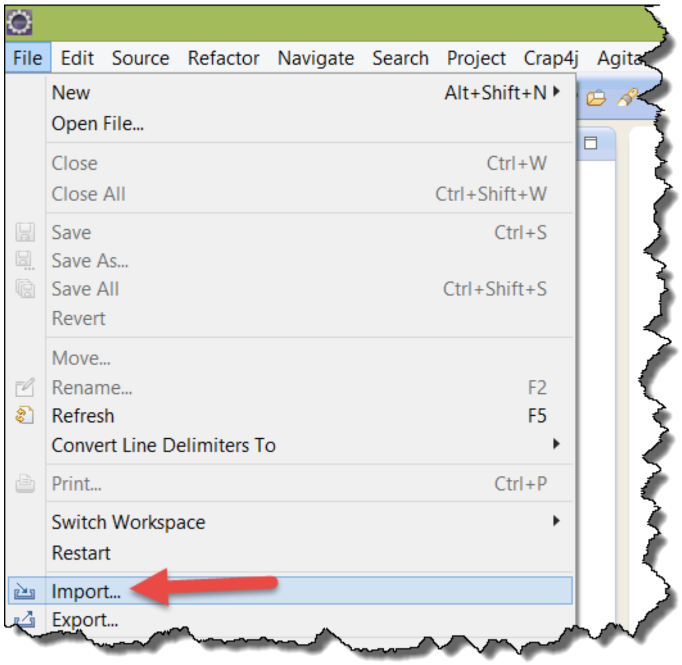
1.1.3 Open your workspace. Type in *C:\Users\<your username >\workspace* and click the *OK* button.


Workspace Launcher
Select a workspace
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.
Workspace: C:\Users\yourname\workspace
Browse...
Use this as the default and do not ask again
OK
Cancel

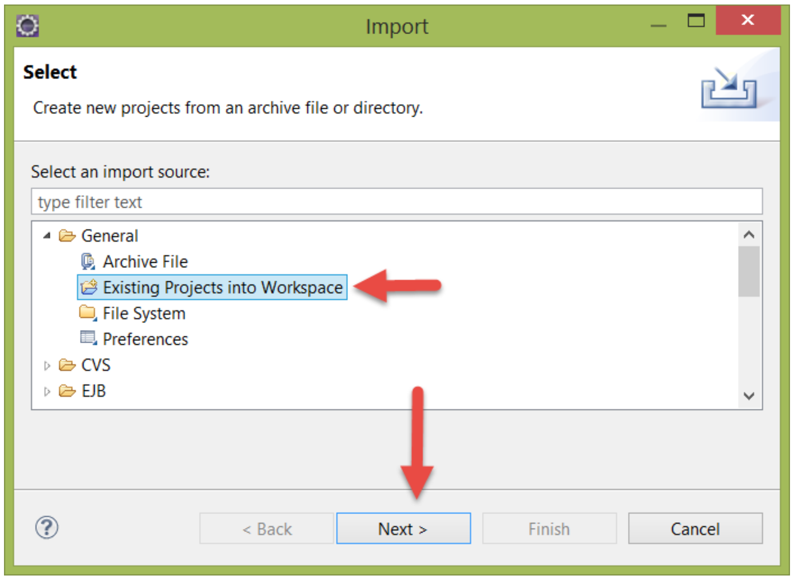
Note: As noted in the past labs, you may use an alternate location for your workspace, but the labs will always reference this location (c:\users\[your username]\workspace) as the default workspace location.

1.2 Import the new Maven Project.

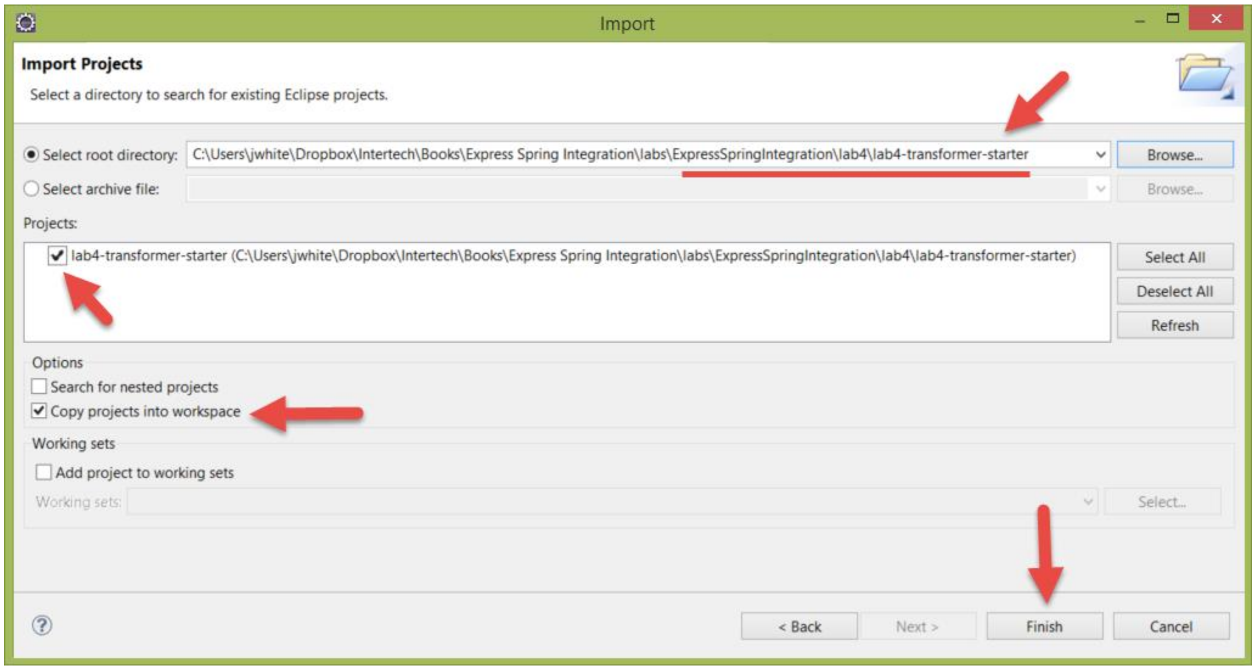
1.2.1 Select File>Import... from the Eclipse menu bar.



1.2.2 Locate and open the General folder in the Import window, and then select *Existing Projects into Workspace Project* from the options. Now push the *Next>* button.



1.2.3 In the "Import" window, use the *Browse…* button to locate the lab4-transformer-starter project folder located in ExpressSpringIntegration\lab4 (this folder is located in the lab downloads). Make sure the project is selected, and the *"Copy projects into workspace"* checkbox is also checked before you hit the *Finish* button (as shown below).



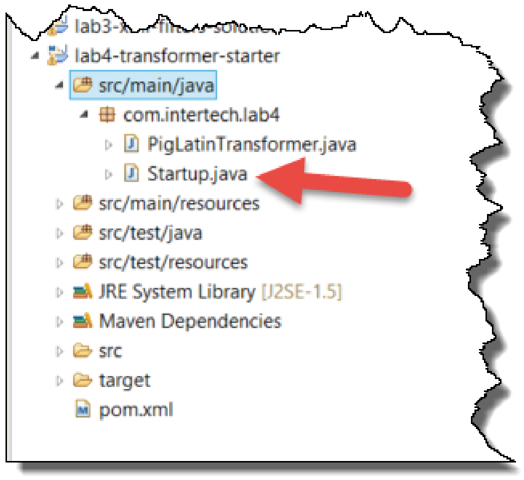
Select root directory: C:\Users\jwhite\Dropbox\Intertech\Books\Express Spring Integration\labs\ExpressSpringIntegration\lab4\lab4-transformer-starter

Projects:
- [✓] lab4-transformer-starter (C:\Users\jwhite\Dropbox\Intertech\Books\Express Spring Integration\labs\ExpressSpringIntegration\lab4\lab4-transformer-starter)

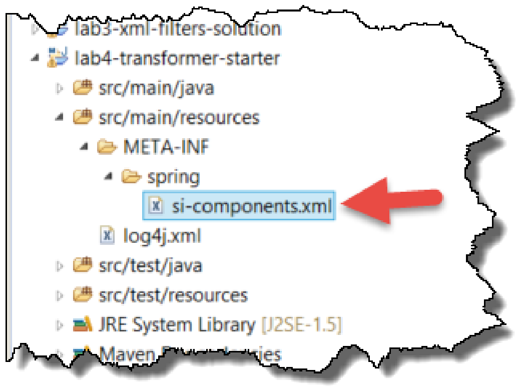
[✓] Copy projects into workspace

1.3 Explore the project. Examine the project for the Spring Integration components that are already present.

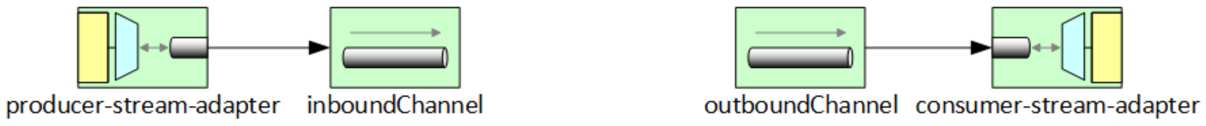
1.3.1 Examine the Startup.java. Expand the src/main/java folder in the Project Explorer view, and open the Startup.java file by double clicking on it. As in past labs, the Startup.java class is used to put the application in an infinite loop so that the SI components can do their work.



1.3.2 Examine the SI configuration. Expand the src/main/resources/META-INF/spring folder in the Project Explorer view, and open si-components.xml file by double clicking on it. The si-components.xml file contains the Spring configuration that includes the definitions for several SI components already.



1.3.3 In particular, note that many of existing components are just like what you saw in Lab 1. There are Standard Input stream and Standard Output stream adapters for reading /writing String text to/from the standard input/output channels. Recall that a Standard Input adapter takes text you enter into the Console view and puts it into a message on a designated message channel – in this case the inboundChannel. The Standard Output adapter takes a message from a channel – in this case the outboundChannel - and displays its contents to the Console view. Below is an EIP model representing what is currently defined in the si-components.xml file.



Note that no connection exists between the inbound and outbound channels at this time.

Note: You may have also noted the existence of a "component-scan" element and the context namespace in the XML file. These will be discussed later in this lab.

Step 2: Create a simple transformer

Spring provides a built-in transformer that uses Spring Expression Language (SpEL) to achieving a simple transformation of the payload without writing a custom transformer. In this example, you configure a simple transformer with a SpEL expression to transform the string contents of the source message into a target message containing the reversed, uppercase string of the source message.

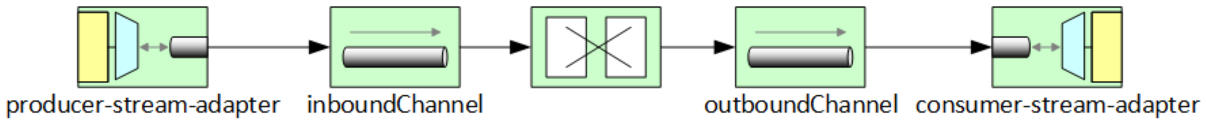
2.1 Add a SI transformer component.

2.1.1 In the same si-components.xml file, add a SI transformer that uses SpEL to reverse and uppercase the string payload of a message from inboundChannel and put it into the outboundChannel.

```
<int:transformer input-channel="inboundChannel"
  output-channel="outboundChannel"
  expression="new
StringBuilder(payload).reverse().toString().toUpperCase()" />
```

Note: if you get stuck or feel like not typing in all the code yourself, you will find a working copy of the final si-component.xml file at ExpressSpringIntegration\lab4\lab4-transformer-solution.

With the addition of this transformer component, you have linked the inbound and outbound channels.



Wolfe and Hohpe call this type of component a "message translator" – but it accomplishes the same goal.

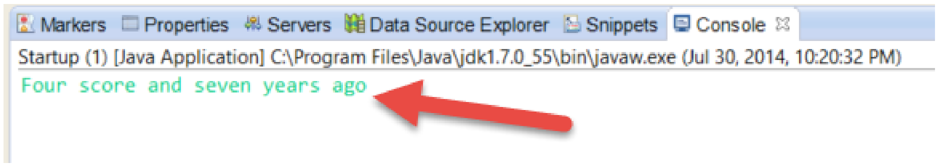
2.1.2 Save the configuration file and make sure there are no errors in the file.

Step 3: Test the Transformer

3.1 Test the application. As you did in lab 1, use the Console view to enter data to the Standard Input stream and allow the transformer to uppercase and reverse your text entry and dump it back out to the Console view (via the Standard Output stream).

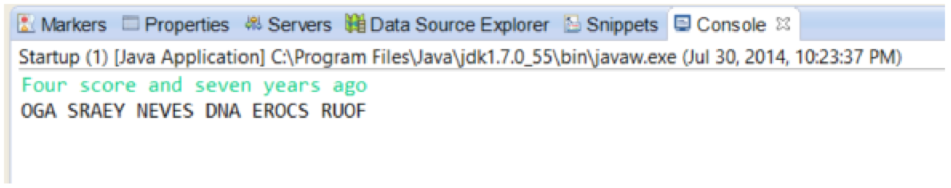
3.1.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

3.1.2 The application is now running awaiting your text input. In the Console view, enter some text and then hit the *Enter* key (by default, your text will be displayed in green).



```
Startup (1) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014, 10:20:32 PM)
Four score and seven years ago
```

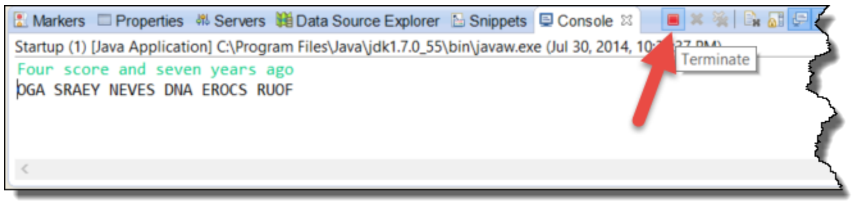
3.1.3 A text message created from the text you enter into the Standard Input where it will immediately be entered into the inboundChannel. Immediately after that, it will be delivered to the transformer which will perform the reverse and uppercase operations on the string (per the SpEL expression). The results will be put in a message on the outboundChannel. The consumer adapter will then display the new message contents back to the Console view (in black).



```
Startup (1) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014, 10:23:37 PM)
Four score and seven years ago
OGA SRAEY NEVES DNA EROCS RUOF
```

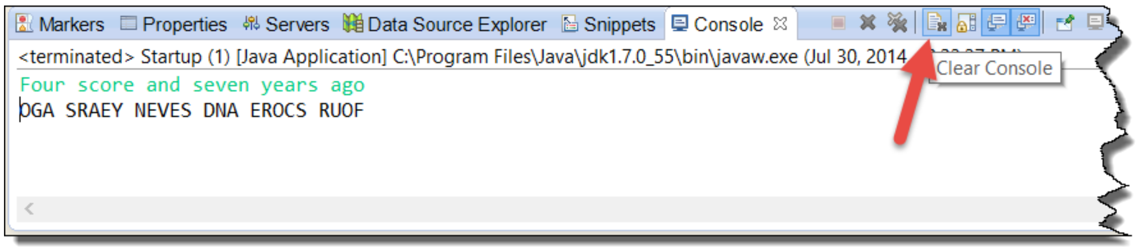
3.2 Stop the application. Recall the application is running in an infinite loop to allow for the constant publishing and consuming of messages. Stop the application now.

3.2.1 In the Console view, click on the red square to terminate the Startup application.



```
Startup (1) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014, 10
Four score and seven years ago
OGA SRAEY NEVES DNA EROCS RUOF
```

3.2.2 The Console view should now indicate that the application is "<terminated>". Clear the Console view by clicking on the Clear Console icon.



```
<terminated> Startup (1) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014
Four score and seven years ago
OGA SRAEY NEVES DNA EROCS RUOF
```

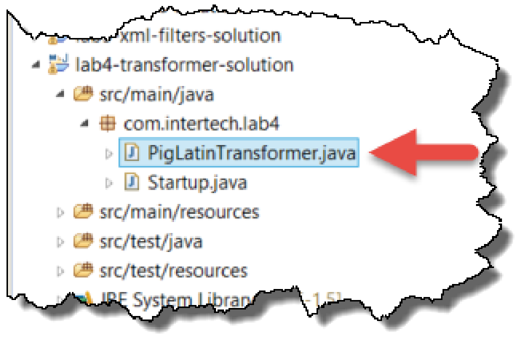
Step 4: Custom Transformer and Spring Annotations

As with filters and many other SI components, you can create your own custom transformer. When the transformation is particularly complex or when you need to transform to / from a type that Spring does not know about, you will find custom transformation the route to take. In this next step, you create a custom transformer – one that transforms the string payload of the source message to a pig Latin translation of the string in the target message. In this step, you also see the use of annotations to configure your transformer. SI (and all of Spring) allows the use of annotations in your Java code to simplify the configuration of components and reduce the amount of XML associated to your application. While annotations were not used in the prior labs (like the filter lab), you will find that SI comes with a number of annotations that can be used in place of XML for the configuration of just about any SI component.

Note: if you are unfamiliar with Pig Latin, you can learn about it here: http://en.wikipedia.org/wiki/Pig_Latin.

4.1 Examine and annotate the Transformer class. A class containing the Pig Latin conversion code has already been created for you. You just need to annotate it in order to designate it as a transformer for SI.

4.1.1 Locate, open, and explore the PigLatinTransformer class in the com.intertech.lab4 package by double clicking on the file in the Package Explorer.



4.1.2 Note that this class has a single method – the toPigLatin( ) method. This message takes in a Message<String> object –the source message - and returns a Message<String> object – the target object. The method grabs the string payload from the message (line 16) and loops through all the words in the String, converting each to a Pig Latin word as it goes (lines 20-45). In the end, it builds and returns a new Message with the translated String as its contents (line 47).

4.2 Annotate the PigLatinTransformer as a Spring component (bean) and as a SI transformer. Annotations provide metadata about the components use and wiring right in the Java code as opposed to the XML configuration.

4.2.1 On top of the class definition, define the PigLatinTransformer as a Spring component or bean.

```
@Component
public class PigLatinTransformer {
  ...
}
```

Any class carrying this annotation on it will be automatically declared a Spring bean in the Spring container just as if you defined the bean with the following declaration.

```
<bean id="pigLatinTransformer"
  class="com.intertech.lab4.PigLatinTransformer"/>
```

The <context:component-scan> element in the si-components.xml file is what tells Spring to go find the beans with these annotations (called Stereotype annotations) and make them beans in the Spring container. In this case, telling it to look or scan the com.intertech.lab4 package for such beans.

```
<context:component-scan base-package="com.intertech.lab4" />
```

4.2.2 Annotate the toPigLatin( ) method with the @Transformer annotation provided by SI.

```
 @Transformer
 public Message<String> toPigLatin(Message<String> inString) {
   ...
 }
```

This SI annotation defines the bean as an SI transformer, and more specifically, defines the toPigLatin method as the method that performs the transformational work. SI will call this method when a message arrives in the inbound message channel. SI takes the resulting return message in the outbound channel.

4.2.3 Add the necessary imports required to use the annotations in the PigLatinTransformer class by hitting Control-Shift-O (all the imports needed in the class – including the new imports - are shown below).

```
import java.util.Scanner;
import org.springframework.integration.annotation.Transformer;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;
```

4.3 Save the class and make sure there are no compile errors in the project.

4.4 Replace the string reverse-uppercase transformer with the Pig Latin transformer.

4.4.1 If not already open, locate the si-components.xml file in src/main/resource/META-INF/spring and open it by double clicking on the file.

4.4.2 Remove the string reverse/uppercase transformer. You can either remove it from the file or comment it out using XML comments as shown below.

```
<!-- <int:transformer input-channel="inboundChannel" -->
<!-- output-channel="outboundChannel" -->
<!-- expression="new
StringBuilder(payload).reverse().toString().toUpperCase()" /> -->
```

4.4.3 Add the Pig Latin Transformer. Note that no bean definition is required for the "ref" attribute. This is because the bean has been declared via annotations (and handled via the component scan).

```
<int:transformer input-channel="inboundChannel"
 output-channel="outboundChannel" ref="pigLatinTransformer" />
```

Again, note that the transformer connects the inboundChannel to the outboundChannel.

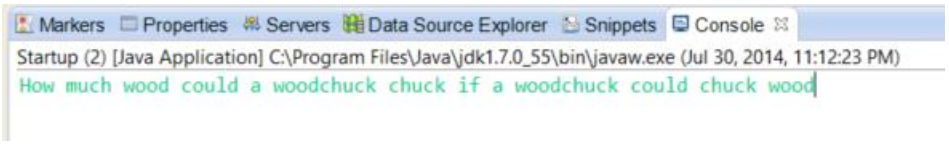
Note: if you get stuck or feel like not typing in all the code yourself, you will find a working copy of the final si-component.xml file at ExpressSpringIntegration\lab4\lab4-transformer-solution\si-component.xml.

4.4.4 Save the configuration file and make sure there are no errors in the file.

4.5 Retest the application (and see the Pig Latin transformation).

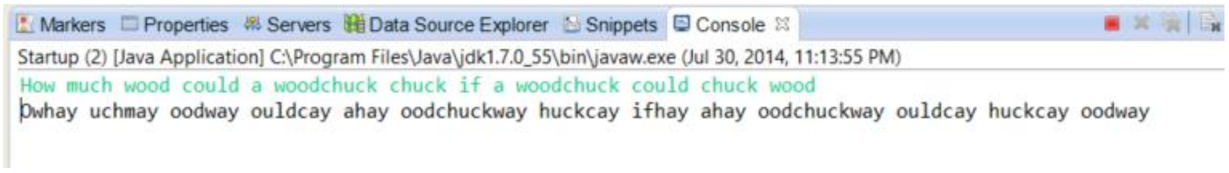
4.5.1 Again locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

4.5.2 Again, enter a text phrase into the Console view and hit enter.



```
Startup (2) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014, 11:12:23 PM)
How much wood could a woodchuck chuck if a woodchuck could chuck wood
```

4.5.3 This time, the custom transformer performs the work of message conversion resulting in the rather unique looking string that displays.



```
Startup (2) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Jul 30, 2014, 11:13:55 PM)
How much wood could a woodchuck chuck if a woodchuck could chuck wood
owhay uchmay oodway ouldcay ahay oodchuckway huckcay ifhay ahay oodchuckway ouldcay huckcay oodway
```

4.5.4 Terminate the application and clear Console view.

Scenario – Transform XML Messages to Java Objects

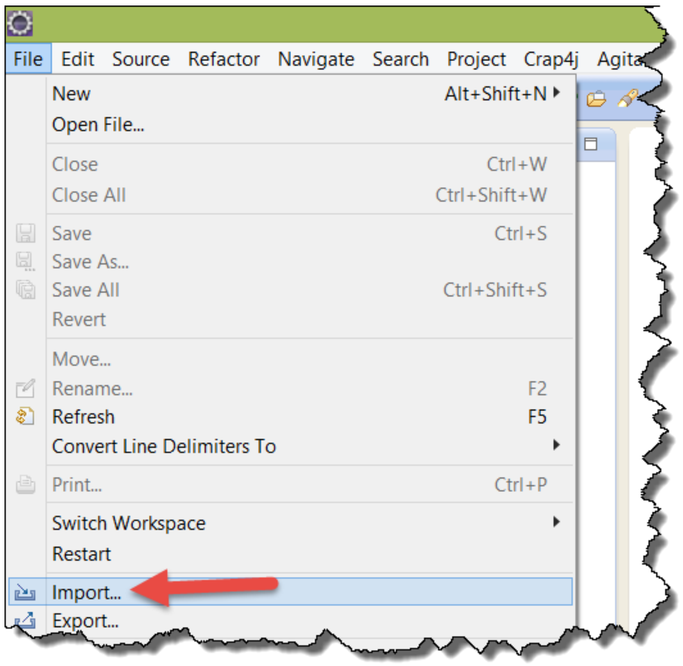
XML is popular way to deliver data. However, Java applications prefer to work with the data in object form. A built-in SI transformer provides the ability to convert an XML payload message into a message containing a Java object holding the data of the XML message. This process is called unmarshalling transformation (an opposite process – going from Java object to XML payload also exists and that is called marshalling). Under the covers, the unmarshalling transformer uses JAXB technology to perform the XML to object work. In this portion of the lab, you explore the use of a SI unmarshalling transformer.

Step 5: Import the Maven Project

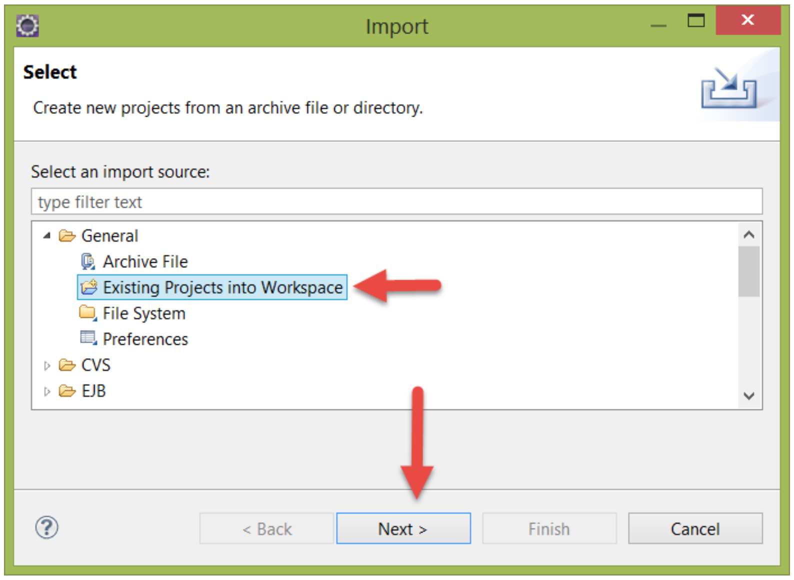
A Maven Project containing the base code for the XML unmarshalling transformer application has already been created for you. You will use this project to begin your exploration of XML unmarshalling transformers.

5.1 Import the new Maven Project.

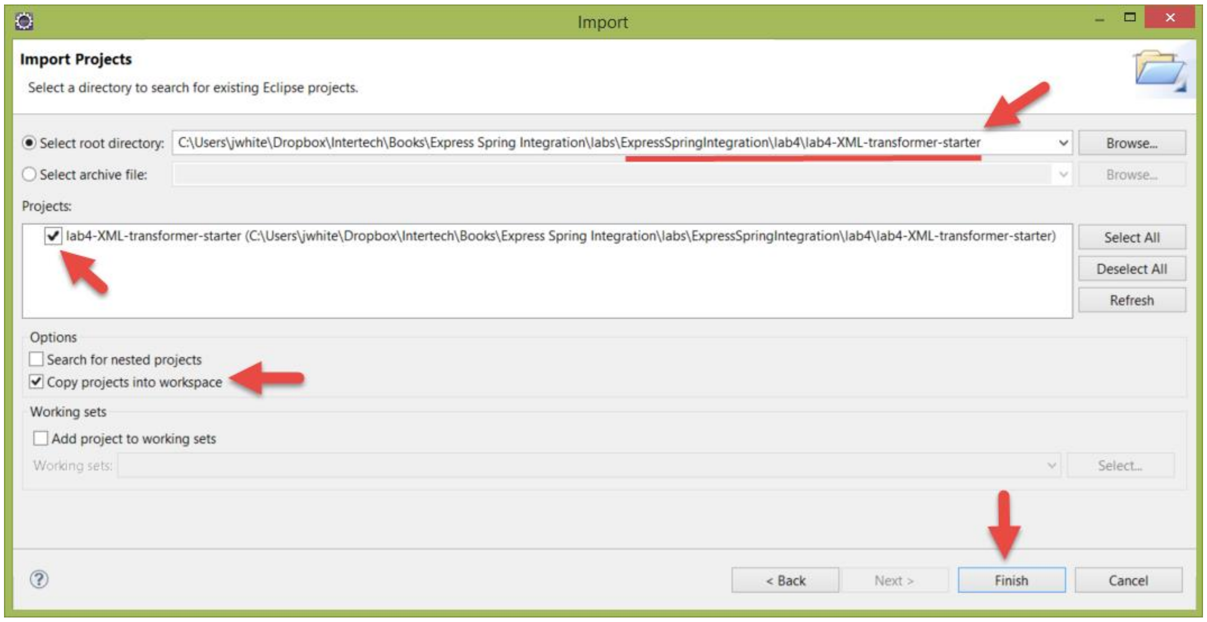
5.1.1 Select File>Import... from the Eclipse menu bar.



5.1.2 Locate and open the General folder in the Import window, and then select *Existing Projects into Workspace Project* from the options. Now push the *Next>* button.



5.1.3 In the "Import" window, use the *Browse…* button to locate the lab4-xml-transformer-starter project folder located in ExpressSpringIntegration\lab4 (this folder is located in the lab downloads). Make sure the project is selected, and the *"Copy projects into workspace"* checkbox is also checked before you hit the *Finish* button (as shown below).



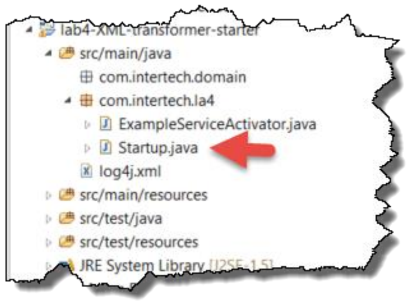
Select root directory: C:\Users\jwhite\Dropbox\Intertech\Books\Express Spring Integration\labs\ExpressSpringIntegration\lab4\lab4-XML-transformer-starter

☑ lab4-XML-transformer-starter (C:\Users\jwhite\Dropbox\Intertech\Books\Express Spring Integration\labs\ExpressSpringIntegration\lab4\lab4-XML-transformer-starter)

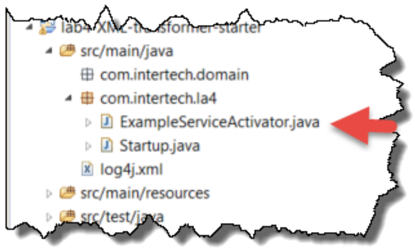
☑ Copy projects into workspace

5.2 Explore the project. Examine the project for the Spring Integration components that are already present.

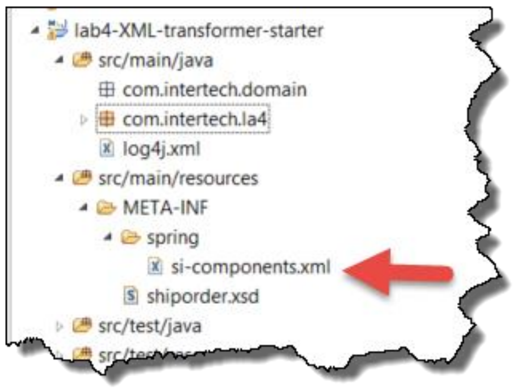
5.2.1 Examine the Startup.java. Expand the src/main/java folder in the Project Explorer view, and open the Startup.java file by double clicking on it. As in past labs, the Startup.java class is used to put the application in an infinite loop so that the SI components can do their work.



5.2.2 Note the presence of another class, ExampleServiceActivator, in the com.intertech.lab4 package. More information about this service activator is below.



5.2.3 Examine the SI configuration. Expand the src/main/resources/META-INF/spring folder in the Project Explorer view, and open si-components.xml file by double clicking on it. The si-components.xml file contains the Spring configuration that includes the definitions for several SI components already.



5.2.4 In particular, note that again, a *file* inbound adapter is already defined in the configuration file (just like in lab 3).

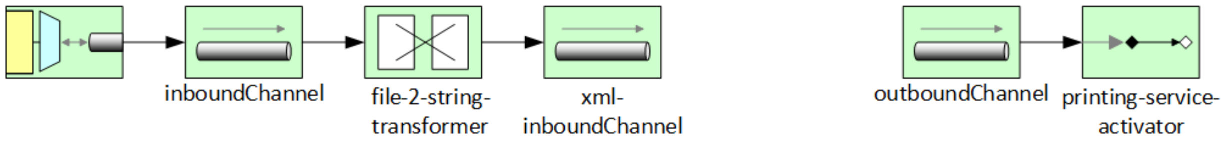
5.2.5 On the end of the outboundChannel sits a service activator (the printing-service-activator). You will learn more about service activators in one of the latter tutorials. What this particular service activator does is to take messages off the associated channel (in this case the outboundChannel that it is hooked to) and print the contents of the message out to the Console view using System.out.println(). You can find the logic associated to the service activator in com.intertech.lab4.ExampleServiceActivator.

```
public class ExampleServiceActivator {

   public void printShiporder(Object order){
          System.out.println(order);
   }
}
```

5.2.6 Ah…, remember the transformer in the lab 3. Its back! Notice the file-to-string-transformer is in place just as it was in lab 3. Now that you know what transformers do, you have a better appreciation of why this SI component is in place. It is what converts the file message into a string message so that the application can deal with the contents of the file versus the file itself.

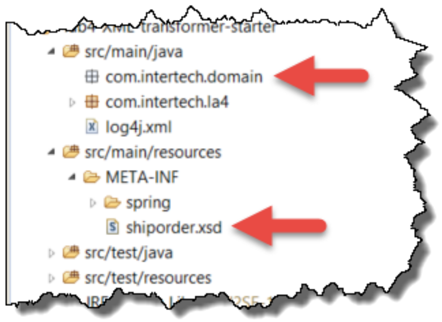
```
<int-file:file-to-string-transformer
  id="file-2-string-transformer" input-channel="inboundChannel"
  output-channel="xml-inboundChannel" charset="UTF-8" />
```

5.2.7 In addition, three channels are defined – as represented in the EIP diagram below.



Note that no connection exists between the xml-inbound and outbound channels at this time.

5.2.8 Finally, note the presence of a shiporder.xsd XML schema file located in the src/main/resources/META-INF folder along with an empty com.intertech.domain package. These will be used to inform the mapping engine to provide the classes needed to receive the data in the XML messages. More on this coming up in the next step.



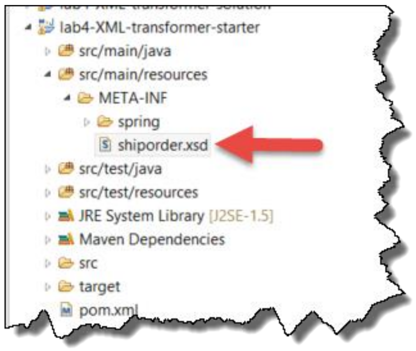
Step 6: Use JAXB to generate the unmarshalling object

SI comes with an out of the box UnmarshallingTransformer to convert XML payloads into objects. Under the covers, the mapping between XML to object is accomplished by your choice of well-known mapping implementations to include JAXB, Castor, and JiBX. As Eclipse provides a built in JAXB tool to generate the needed Java classes from schema, you will use the JAXB mapping implementation for the unmarshalling transformation for this step in the lab.

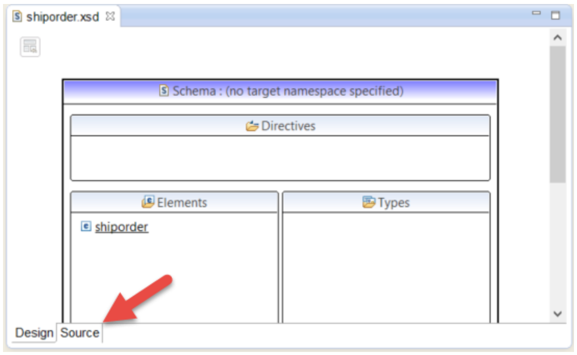
Note: SI also comes with an out of the box support for marshalling objects to XML as well through a MarshallingTransformer component that can also use the mapping technologies listed above.

6.1 Generate Java Classes from a schema. In this portion of the lab, your SI application will accept a number of shipment order messages in XML. The XML messages will be transformed into Shiporder objects by the SI UnmarshallingTransformer. You could create the target Java classes by hand, but that usually proves tedious and error prone. Most XML to object mapping technology provides class generation tools to create the needed Java classes. Eclipse comes with built in JAXB support to generate Java classes from an XML schema that defines the incoming messages. Use that tool now to generate the Java classes from the shiporder.xsd.

6.1.1 Locate the shiporder.xsd file in src/main/resource/META-INF/ folder and open it by double clicking on the file.

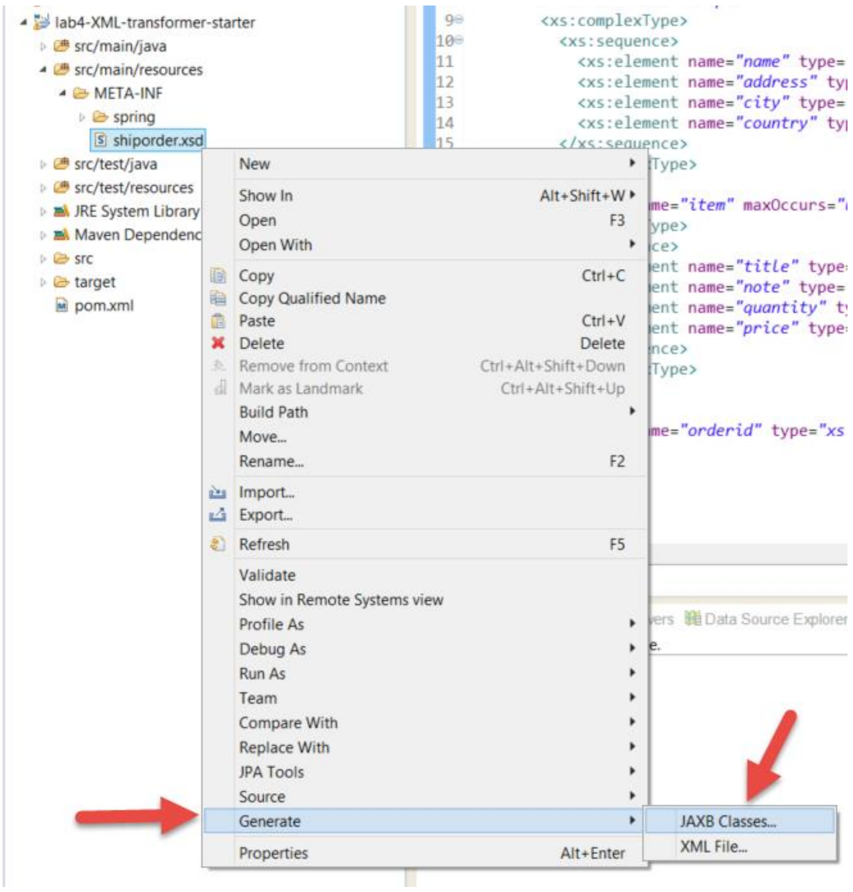


6.1.2 The file will probably open in the Design view. Use the tabs at the bottom of the view to change to the Source view to see the source XML for the schema.



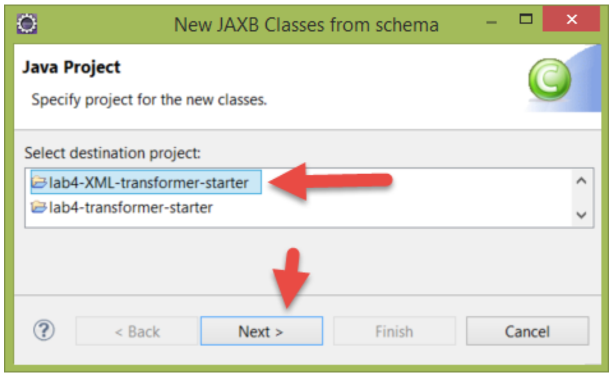
6.1.3 Explore the schema. Note the definition of the shiporder, orderperson, and item elements in particular. This schema defines the incoming XML shipment orders that your SI application will handle. Messages of this ilk will need to be transformed ("unmarshalled") into Java objects by the SI transformer. Before the transformer can do its work, your application needs classes that define the objects that the transformer to create. Eclipse has a built in JAXB tool that can use this file to generate those classes.

6.1.4 Right click on the shiporder.xsd file in the Project Explorer view and select Generate > JAXB Classes… from the resulting menu.



```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type=
        <xs:element name="address" ty
        <xs:element name="city" type=
        <xs:element name="country" ty
      </xs:sequence>
```

6.1.5 In the New JAXB Classes from schema window that appears, select lab4-XML-transformer-starter as the project for the new classes and press the Next> button.


New JAXB Classes from schema

Java Project

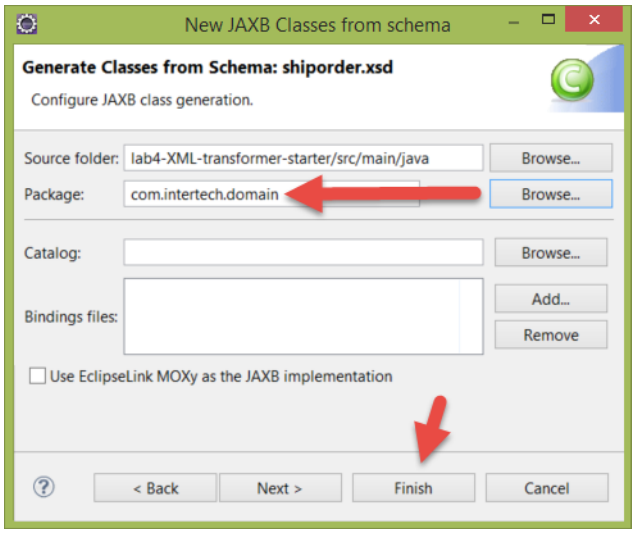
Specify project for the new classes.

Select destination project:

lab4-XML-transformer-starter

lab4-transformer-starter

6.1.6 In the next window, use the Browse... button next to the Package field to select the com.intertech.domain package. This specifies where the new Java classes will be placed in the project. Leave the other fields blank as shown below and press the Finish button.


New JAXB Classes from schema

Generate Classes from Schema: shiporder.xsd

Configure JAXB class generation.

Source folder: lab4-XML-transformer-starter/src/main/java

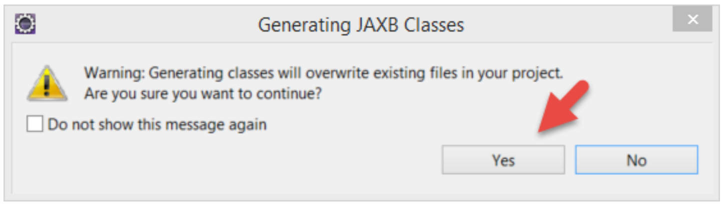
Package: com.intertech.domain

Catalog:

Bindings files:

Use EclipseLink MOXy as the JAXB implementation

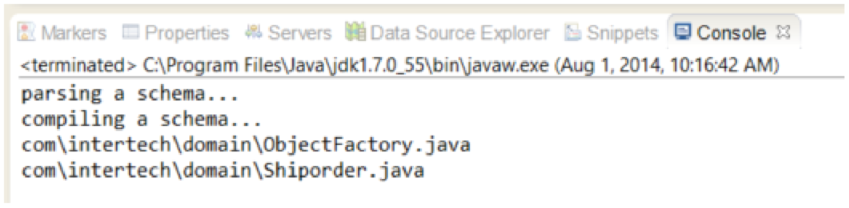
6.1.7 Acknowledge the warning about overwriting existing files by pressing Yes on the next window.


Generating JAXB Classes

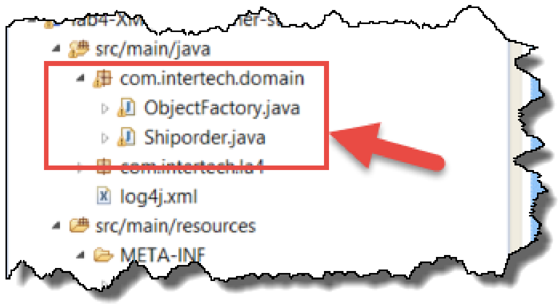
Warning: Generating classes will overwrite existing files in your project.
Are you sure you want to continue?

Do not show this message again

6.1.8 If you watch the Console view, Eclipse will inform you of the work it is perform. When complete, you should find new classes in the com.intertech.domain package.

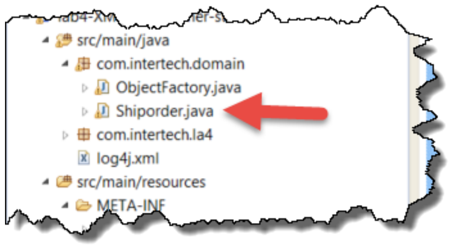


```
<terminated> C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Aug 1, 2014, 10:16:42 AM)
parsing a schema...
compiling a schema...
com\intertech\domain\ObjectFactory.java
com\intertech\domain\Shiporder.java
```

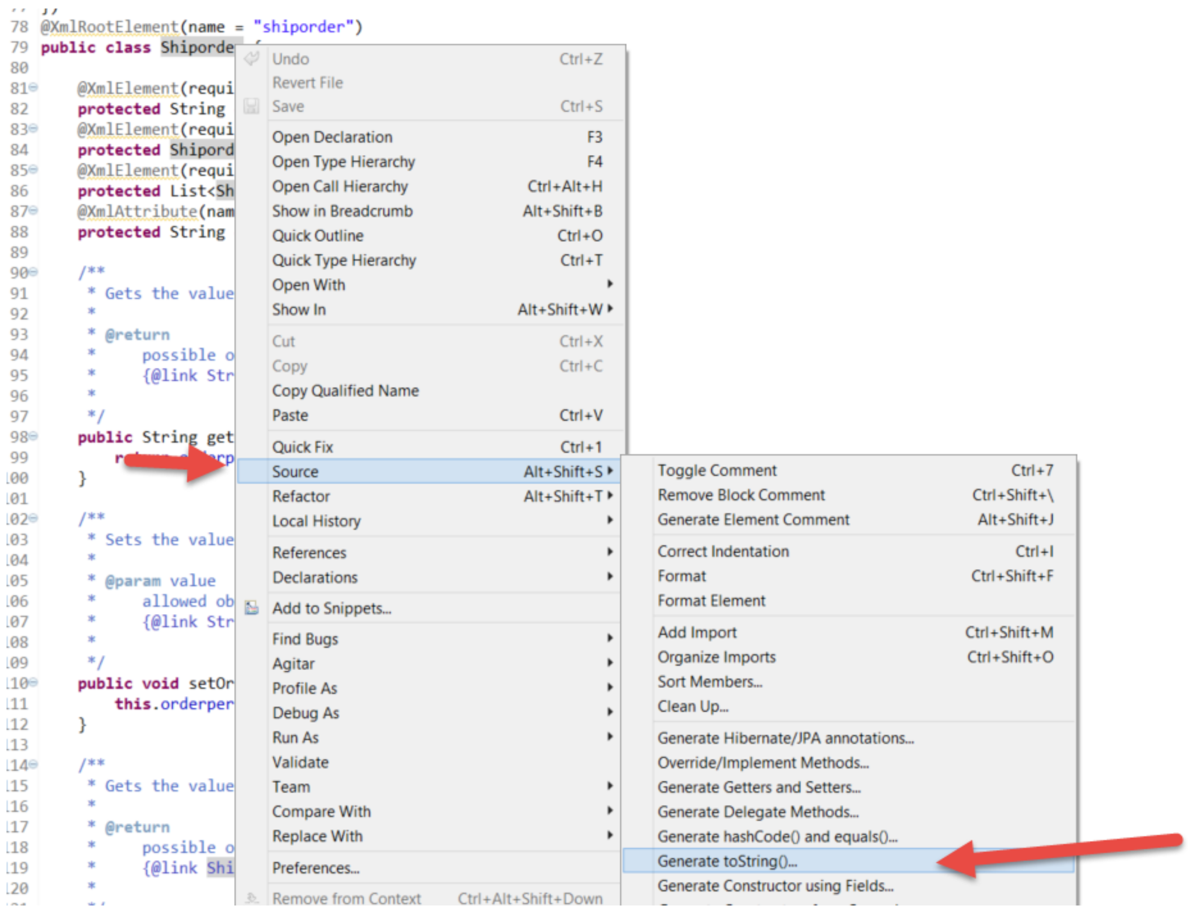


6.2 Add toString( ) methods to the JAXB generated classes. In order to see the results of the SI transformer best, it is helpful to put toString( ) methods on all of the JAXB generated classes. The toString( ) method can display the contents of the objects and will be used by the service activator to display the contents to the Console View.

6.2.1 Locate and open the newly created Shiporder.java class found in the com.intertech.domain package in src/main/java folder by double clicking on the file in the Project Explorer.

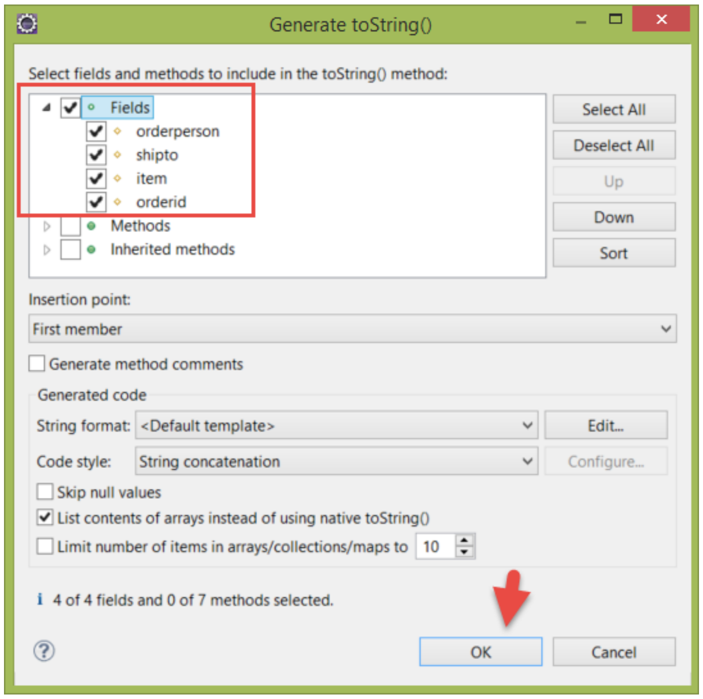


6.2.2 In this file, there are three classes defined: Shiporder and two inner classes Item and Shipto (it might be difficult to find them given all the commenting that the JAXB generator added – feel free to delete the comments if it helps). Allow Eclipse to generate toString( ) methods for each of these three classes. Right click on each of the three classes in the editor and select Source > Generate toString()… from the resulting menu as shown below.



```
@XmlRootElement(name = "shiporder")
public class Shiporde

    @XmlElement(requi
    protected String
    @XmlElement(requi
    protected Shipord
    @XmlElement(requi
    protected List<Sh
    @XmlAttribute(nam
    protected String

    /**
     * Gets the value
     *
     * @return
     *     possible o
     *     {@link Str
     *
     */
    public String get
        return orderp
    }

    /**
     * Sets the value
     *
     * @param value
     *     allowed ob
     *     {@link Str
     *
     */
    public void setOr
        this.orderper
    }

    /**
     * Gets the value
     *
     * @return
     *     possible o
     *     {@link Shi
     *
```

6.2.3 In the resulting Generate toString( ) window, make sure all the fields for each class are selected and then press the OK button (shown below for the Shiporder class). Don't forget to do this for each of the three classes (Shiporder, Item, and Shipto).



Select fields and methods to include in the toString() method:

- [✓] Fields
  - [✓] orderperson
  - [✓] shipto
  - [✓] item
  - [✓] orderid
- [ ] Methods
- [ ] Inherited methods

Insertion point: First member

[ ] Generate method comments

Generated code

String format: <Default template>

Code style: String concatenation

[ ] Skip null values

[✓] List contents of arrays instead of using native toString()

[ ] Limit number of items in arrays/collections/maps to 10

4 of 4 fields and 0 of 7 methods selected.

6.3 Save the Shiporder class and make sure there are no compile errors in the project.

Step 7: Add the XML Unmarshalling Transformer

With the target object types now generated, it is time to add the SI components to perform the XML to object transformation.

7.1 Add the JAXB marshalling/unmarshalling bean. Again, the SI transformer works with any number of XML to object mapping technologies. You are going to use a JAXB mapping object (provided with Spring Integration) to do the mapping work under the covers for the transformer.

7.1.1 Locate the si-components.xml file in src/main/resource/META-INF/spring and open it by double clicking on the file.

7.1.2 Add a Spring bean of the Jaxb2Marshaller type, as shown below, into the configuration. This bean can do both unmarshalling and marshalling work.

```
<bean id="jaxbMarshaller"
    class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
  <property name="contextPath" value="com.intertech.domain" />
</bean>
```

7.2 Add a SI transformer component.

7.2.1 In the same si-components.xml file, add a SI unmarshalling-transformer that takes a message from the inbound channel, transforms it to an object (using the JAXB marshaller) and puts the new object message in the outbound channel.

```
<int-xml:unmarshalling-transformer
  id="xml-2-object-transformer" input-channel="xml-inboundChannel"
  output-channel="outboundChannel" unmarshaller="jaxbMarshaller" />
```

Note that the transformer references the unmarshalling bean.

Note: if you get stuck or feel like not typing in all the code yourself, you will find a working copy of the final si-component.xml file at ExpressSpringIntegration\lab4\lab4-XML-transformer-solution folder.

7.2.2 Save the configuration file and make sure there are no errors in the file.

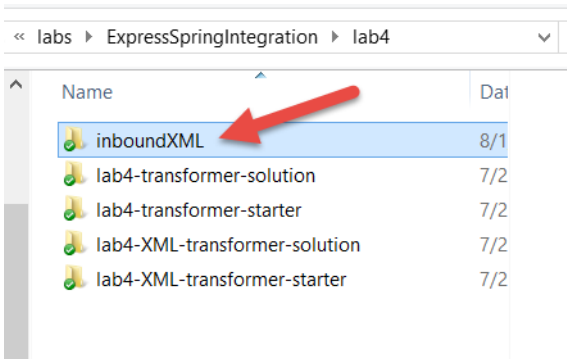
Step 8: Test the Unmarshalling Filter

Add messages into the inbound file folder and then test the application to see the unmarshalling transformer do its job.

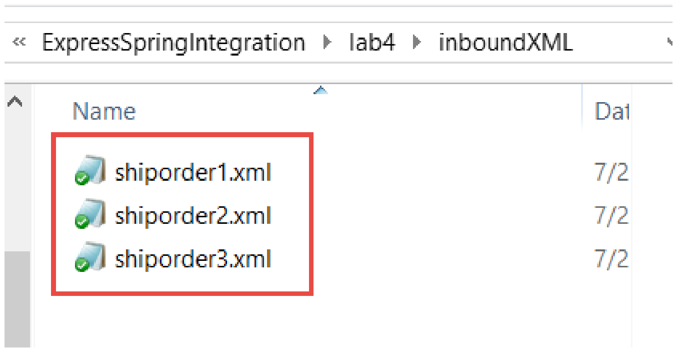
8.1 Add the files to the inbound message folder.

8.1.1 In the si-components.xml, find the producer-file-adapter. Note the location of the directory. It is set, by default, to file:c://inboundXML. This is the location where XML shiporder messages will be taken into the application by the adapter. If it does not already exist for the last lab, create this message folder - changing the location to suit your needs and your file system (change the producer-file-adapter to reflect your location).

8.1.2 Some sample XML shiporder messages have been provided to you. Find them in an inboundXML folder in ExpressSpringIntegration\lab4.


« labs ▸ ExpressSpringIntegration ▸ lab4
Name
Dat
inboundXML 8/1
lab4-transformer-solution 7/2
lab4-transformer-starter 7/2
lab4-XML-transformer-solution 7/2
lab4-XML-transformer-starter 7/2

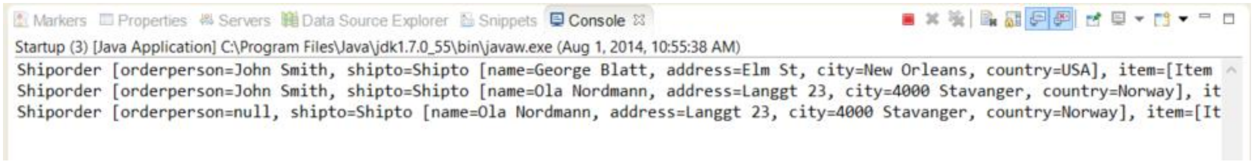
If you open the 3 messages in the folder, you will note that each is a shipment order following the shiporder.xsd schema. Copy the messages from this folder to the c:\\inboundXML folder (or whatever folder you created per 8.1.1. above).



8.2 Test the application. Test the application to see the files in the inbound file folder are transformed to objects and then displayed via their toString( ) methods to the Console view by the ExampleServiceActivator.

8.2.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

8.2.2 Give the application a few seconds to digest each message. Going from XML to object takes a bit more work than reversing a string. Each XML message should eventually show up in the Console view as the toString( ) representation of the object that was created by the transformer.



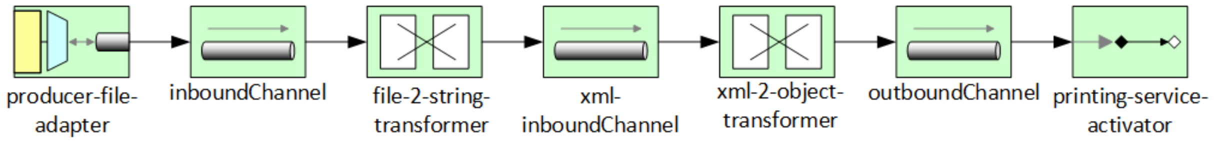
```
Startup (3) [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.exe (Aug 1, 2014, 10:55:38 AM)
Shiporder [orderperson=John Smith, shipto=Shipto [name=George Blatt, address=Elm St, city=New Orleans, country=USA], item=[Item
Shiporder [orderperson=John Smith, shipto=Shipto [name=Ola Nordmann, address=Langgt 23, city=4000 Stavanger, country=Norway], it
Shiporder [orderperson=null, shipto=Shipto [name=Ola Nordmann, address=Langgt 23, city=4000 Stavanger, country=Norway], item=[It
```

8.3 Stop the application. Recall the application is running in an infinite loop to allow for the constant publishing and consuming of messages. Stop the application now.

8.3.1 In the Console view, click on the red square to terminate the Startup application.

8.3.2 The Console view should now indicate that the application is "<terminated>". Clear the Console view by clicking on the Clear Console icon.

8.3.3 Below is the EIP model for your completed unmarshalling transformer application.



Lab Solution

si-components.xml - for the string transformer labs

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int-
stream="http://www.springframework.org/schema/integration/stream"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-
integration.xsd
  http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <!-- Scans within the base package of the application for
@Components to
    configure as beans -->
  <context:component-scan base-package="com.intertech.lab4" />

  <!-- message producer / a Spring Integration wrapped Java Standard
input
    stream -->
  <int-stream:stdin-channel-adapter id="producer-stream-adapter"
    channel="inboundChannel" />

  <int:channel id="inboundChannel" />

  <!-- <int:transformer input-channel="inboundChannel" -->
  <!-- output-channel="outboundChannel" -->
  <!-- expression="new
StringBuilder(payload).reverse().toString().toUpperCase()"
    /> -->

  <int:transformer input-channel="inboundChannel"
    output-channel="outboundChannel" ref="pigLatinTransformer" />

  <int:channel id="outboundChannel" />

  <int-stream:stdout-channel-adapter
    id="consumer-stream-adapter" channel="outboundChannel" append-
newline="true" />

  <int:poller id="defaultPoller" default="true"
    max-messages-per-poll="5" fixed-rate="200" />

</beans>
```

PigLatinTransformer.java

```java
package com.intertech.lab4;

import java.util.Scanner;
import org.springframework.integration.annotation.Transformer;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

@Component
public class PigLatinTransformer {

  @Transformer
  public Message<String> toPigLatin(Message<String> inString) {
    String word;
    String latin = "";
    StringBuilder latinPhrase = new StringBuilder();
    char first;
    boolean cap = false;
    String line = inString.getPayload();
    Scanner pig = new Scanner(line);

    // loop through all the words in the line
    while (pig.hasNext()) // is there another word?
    {
      word = pig.next();
      first = word.charAt(0);
      if ('A' <= first && first <= 'Z') // first is capital letter
      {
        first = Character.toLowerCase(first);
        cap = true;
      } else
        cap = false;

      // test if first letter is a vowel
      if (first == 'a' || first == 'e' || first == 'i' || first == 'o'
          || first == 'u')
        latin = word + "hay";
      else // not a vowel
      {
        if (cap) {
          latin = "" + Character.toUpperCase(word.charAt(1));
          latin = latin + word.substring(2) + first + "ay";
        } else
          latin = word.substring(1) + first + "ay";
      }
      latinPhrase.append(latin + " ");

    }
    pig.close();
    return MessageBuilder.withPayload(latinPhrase.toString()).build();
  }
}
```

si-components.xml – for the XML unmarshalling transformer lab

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-
file="http://www.springframework.org/schema/integration/file"
  xmlns:int-
mail="http://www.springframework.org/schema/integration/mail"
  xmlns:int-
xml="http://www.springframework.org/schema/integration/xml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int-
stream="http://www.springframework.org/schema/integration/stream"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-
integration.xsd
  http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd
  http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/spring-
integration-file.xsd
  http://www.springframework.org/schema/integration/xml
http://www.springframework.org/schema/integration/xml/spring-
integration-xml.xsd">

  <!-- Adapter for reading files -->

  <int-file:inbound-channel-adapter id="producer-file-adapter"
    channel="inboundChannel" directory="file:c://inboundXML"
    prevent-duplicates="true">
    <int:poller fixed-rate="5000" />
  </int-file:inbound-channel-adapter>

  <int:channel id="inboundChannel" />

  <int-file:file-to-string-transformer
    id="file-2-string-transformer" input-channel="inboundChannel"
    output-channel="xml-inboundChannel" charset="UTF-8" />

  <int:channel id="xml-inboundChannel" />

  <int-xml:unmarshalling-transformer
    id="xml-2-object-transformer" input-channel="xml-inboundChannel"
    output-channel="outboundChannel" unmarshaller="jaxbMarshaller" />

  <bean id="jaxbMarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="contextPath" value="com.intertech.domain" />
  </bean>

  <int:channel id="outboundChannel" />
  <int:service-activator id="printing-service-activator"
    input-channel="outboundChannel" ref="serviceActivator" />
  <bean id="serviceActivator"
class="com.intertech.lab4.ExampleServiceActivator" />

</beans>
```