Lab Exercise

Understanding Channels - Lab 1

Channels are an integral part of any Spring Integration application. There are many channels to choose from. Understanding the basic channel types (subscribable and pollable) is essential to any Spring Integration developer. In this lab, you explore the basic channel types and see the default channel at work.

Specifically, in this lab you will:

- Create a new Spring Integration Maven Project
- Define the Spring configuration file.
- Define a Java SE class for running/testing your Spring Integration components.
- Create and test a subscribable channel.
- Create and test a pollable channel.
- Create and test the default Spring Integration channel – the direct channel.

Lab solution folder: ExpressSpringIntegration\lab1\lab1-channels-solution

Scenario

In this lab, you create your first Spring Integration (SI) project. Specifically, you will create a standard Maven project. You will use Maven to bring the necessary Spring libraries into the project. You also create a small SI application that uses several channel types in order to understand the various types of SI channels.

Step 1: Create a Maven Project

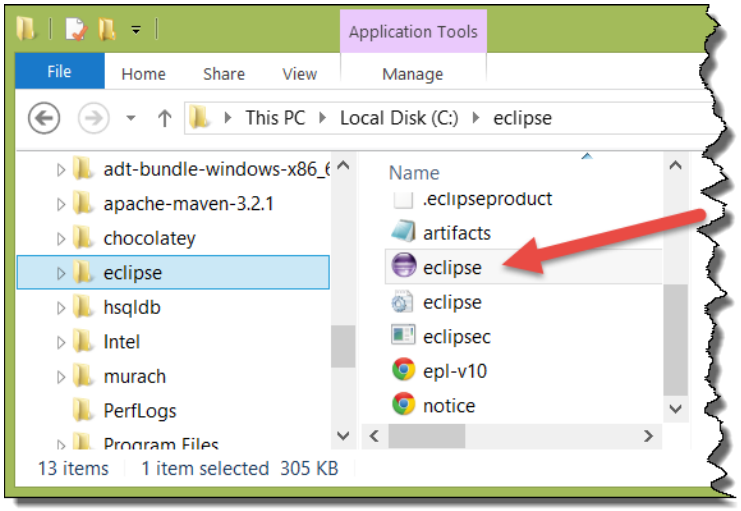
Maven is often used today to easily manage the dependencies of a Java project. Eclipse provides a "Maven Project" for creating a Maven driven/managed Java application in Eclipse (or Eclipsed based IDEs like IBM's Rational Application Developer or the Spring Tool Suite). In this step, you create a simple Maven project and configure the pom.xml to setup the required dependencies (namely Spring dependencies) for use in your project.

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

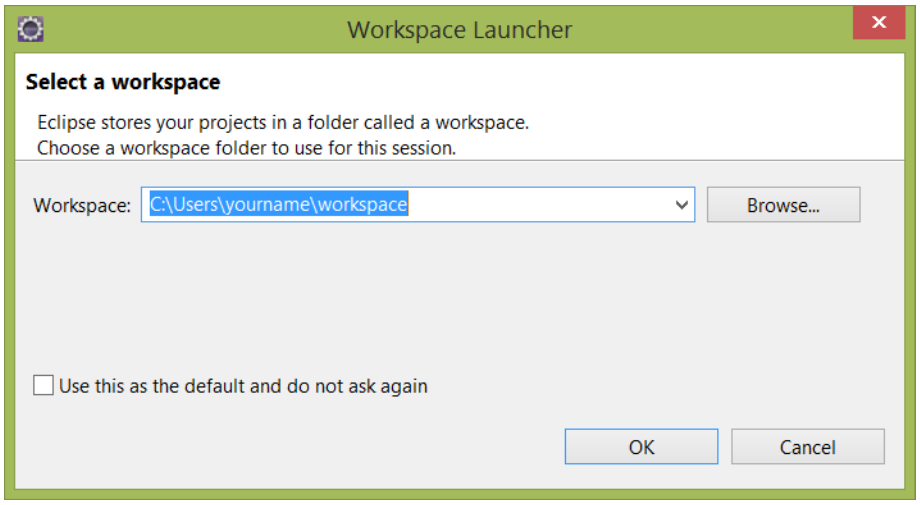
1.1.1 Locate the Eclipse folder.

Note: In Intertech classrooms, Eclipse will be installed in a C:\eclipse folder.

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



1.1.3 Create your workspace. When Eclipse launches, you will be asked to select a workspace. This is the location where your work will be stored. Type in *C:\Users\<your username >\workspace*.


Workspace Launcher

Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

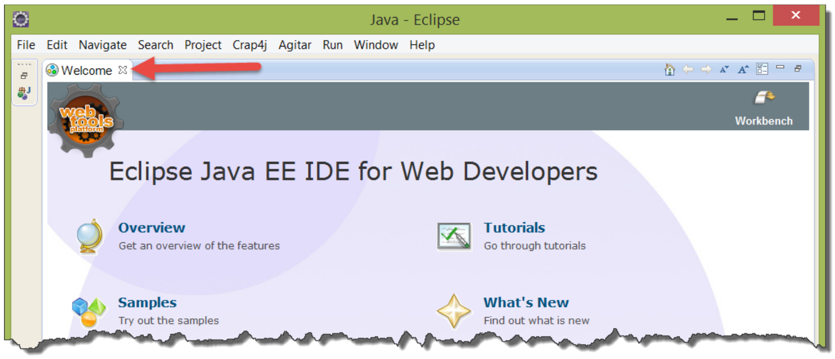
Workspace: C:\Users\yourname\workspace

☐ Use this as the default and do not ask again

Note: 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.

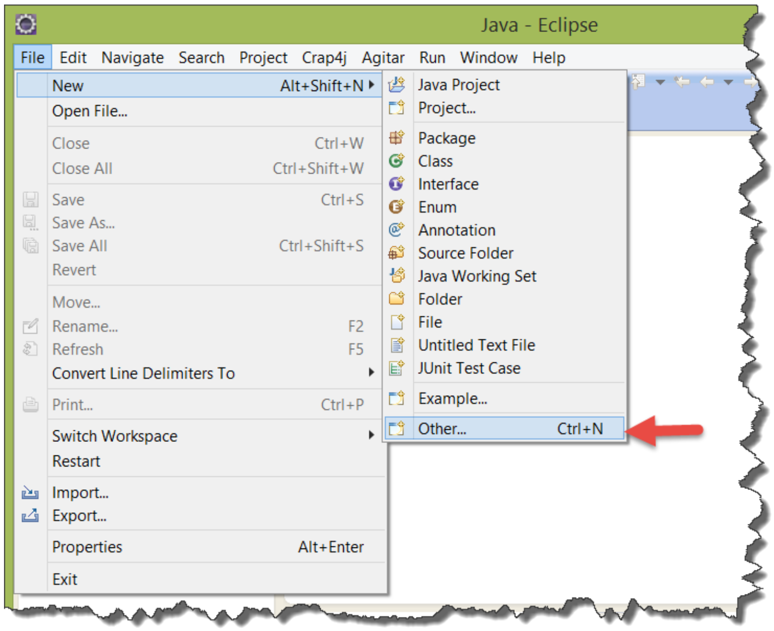
Hit the ***OK*** button. After clicking the OK button, Eclipse will open.

1.1.4 Close the Welcome tab. When Eclipse opens, close the welcome tab by clicking on the X on the tab.

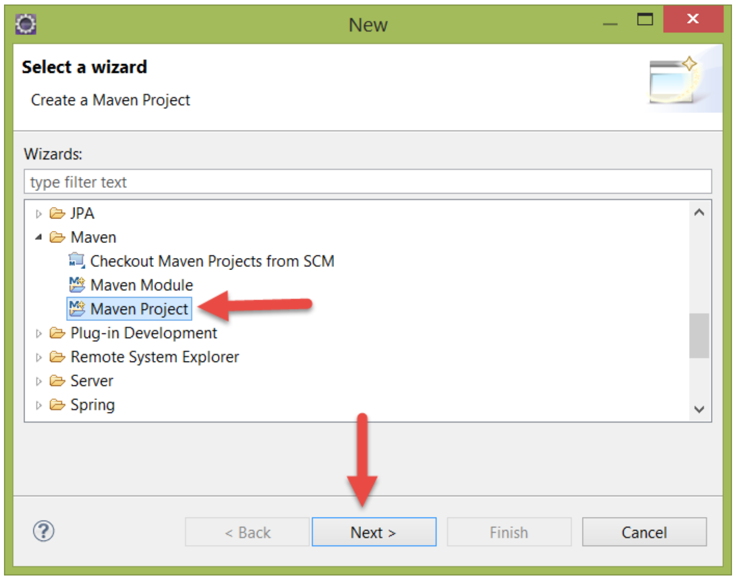

Java - Eclipse
File Edit Navigate Search Project Crap4j Agitar Run Window Help
Welcome
Workbench
Eclipse Java EE IDE for Web Developers
Overview
Get an overview of the features
Tutorials
Go through tutorials
Samples
Try out the samples
What's New
Find out what is new

1.2 Create a new Maven Project. Create a new Eclipse project for this lab work.

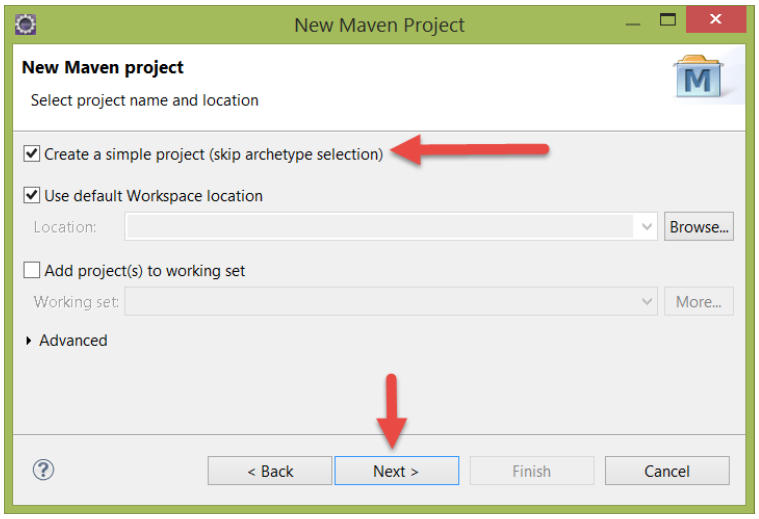
1.2.1 Select *File>New>Other...* from the Eclipse menu bar.



1.2.2 Locate and open the Maven folder in the New, "Select a wizard" window, and then select *Maven Project* from the options. Now push the *Next>* button.



1.2.3 In the "New Maven Project" window, check *the Create a simple project (skip archetype selection)* and then push the *Next>* button.


New Maven project

Select project name and location

☑ Create a simple project (skip archetype selection)

☑ Use default Workspace location

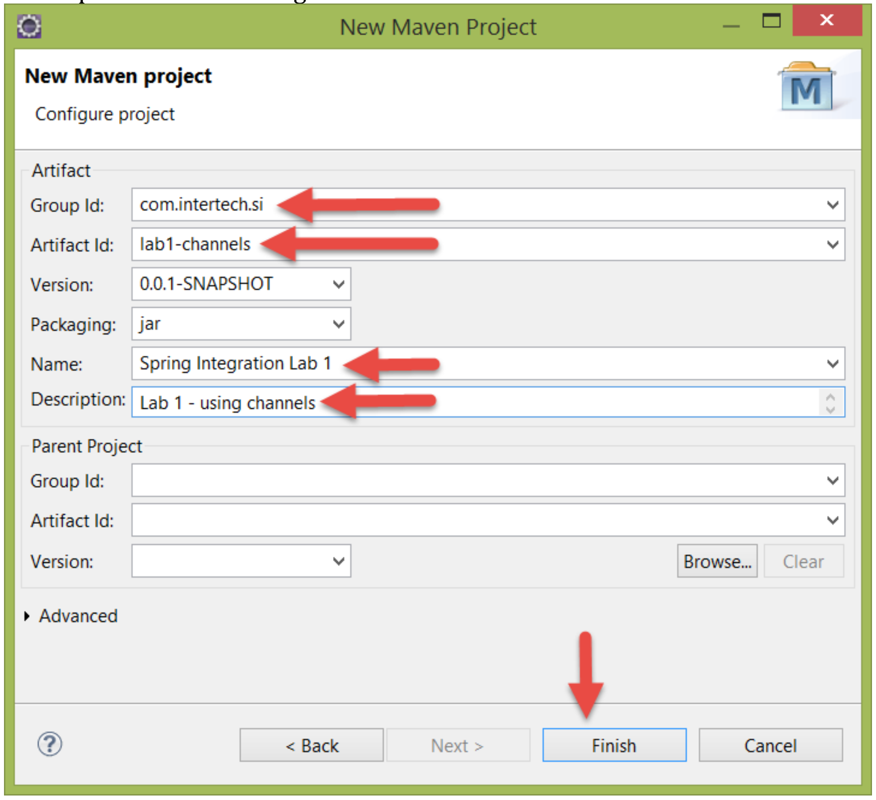
Location:

☐ Add project(s) to working set

Working set:

1.2.4 In the next window, enter the following values for the Artifact entries (as shown in the picture below) while leaving the default values for version and packaging, and hit the *Finish* button:

- Group Id: com.intertech.si
- Artifact Id: lab1-channels
- Name: Spring Integration Lab 1
- Description: Lab 1 – using channels


New Maven project

Configure project

Artifact

Group Id: com.intertech.si

Artifact Id: lab1-channels

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name: Spring Integration Lab 1

Description: Lab 1 - using channels

Parent Project

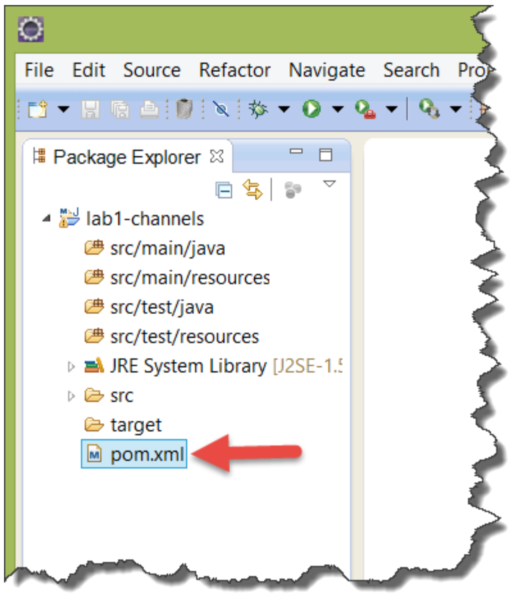
Group Id:

Artifact Id:

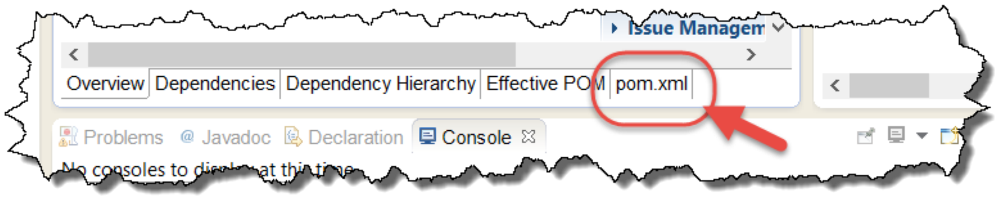
Version:

1.3 Add Spring Integration dependencies. Your project needs Spring Integration, Spring Framework and other libraries. Add the necessary dependencies to the Maven pom.xml file in order to add the necessary JAR files to your local repository and to the project build/execution paths.

1.3.1 Expand the newly created lab1-channels project in the Package Explorer view. Locate and open the pom.xml file in an editor by double clicking on it.



1.3.2 Push on the pom.xml tab in the editor options to open the pom.xml in an XML editor.



1.3.3 Add the following properties and dependencies to the pom.xml. The properties and dependencies should be inserted after the <description> element in the XML file.

Note: this code can be copied from the solutions download at ExpressSpringIntegration\lab1\lab1-channels-helper-code\pom.xml if you do not wish to enter the text by hand.

```
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<springframework.version>4.0.4.RELEASE</springframework.version>
<spring.integration.version>4.0.0.RELEASE</spring.integration.version>
<spring.integration.xml.version>4.0.0.RELEASE</spring.integration.xml.
version>
<log4j.version>1.2.17</log4j.version>
</properties>

  <dependencies>

    <dependency>
      <groupId>org.springframework.integration</groupId>
      <artifactId>spring-integration-core</artifactId>
      <version>${spring.integration.version}</version>
    </dependency>

    <dependency>
      <groupId>org.springframework.integration</groupId>
      <artifactId>spring-integration-stream</artifactId>
      <version>${spring.integration.version}</version>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>${log4j.version}</version>
    </dependency>

  </dependencies>
```

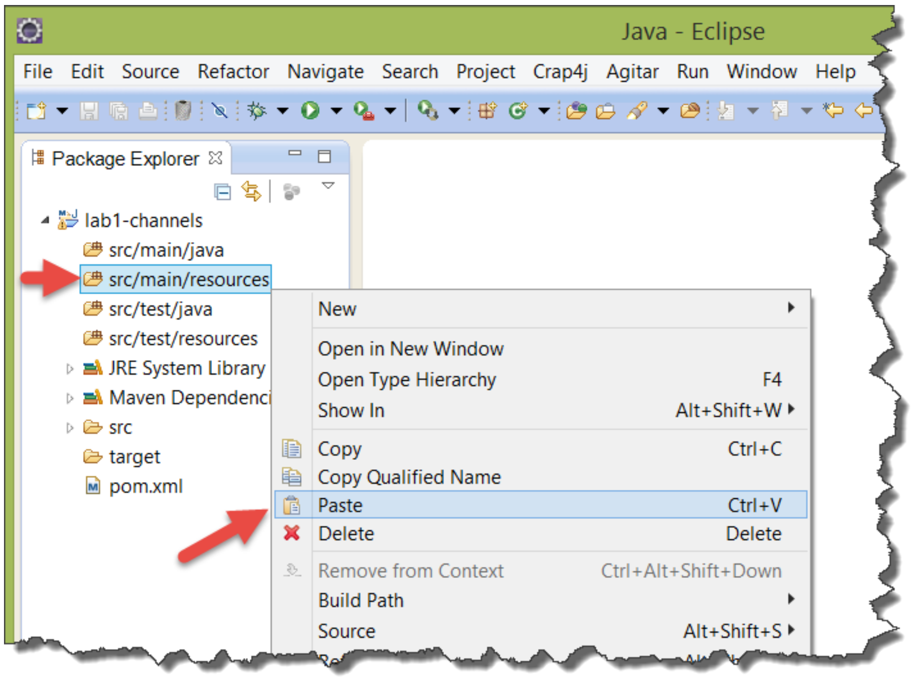
Note that the dependency list includes Spring Integration (version 4) and Spring Integration Stream, and Log4J.

1.3.4 Save the pom.xml file and make sure there are no errors in the project.

1.4 Copy the log4j configuration to the project. Your application (and Spring Integration) will use Log4J for all application logging.

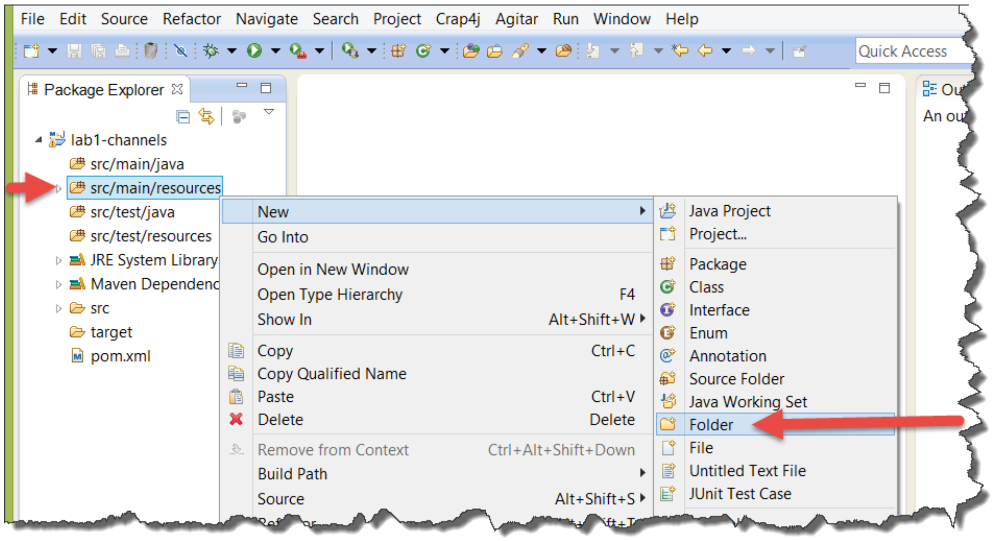
1.4.1 Locate the log4j.xml file located in ExpressSpringIntegration\lab1\lab1-channels-helpler-code.

1.4.2 Copy the file and paste the file in the src/main/resources folder of your Eclipse project.

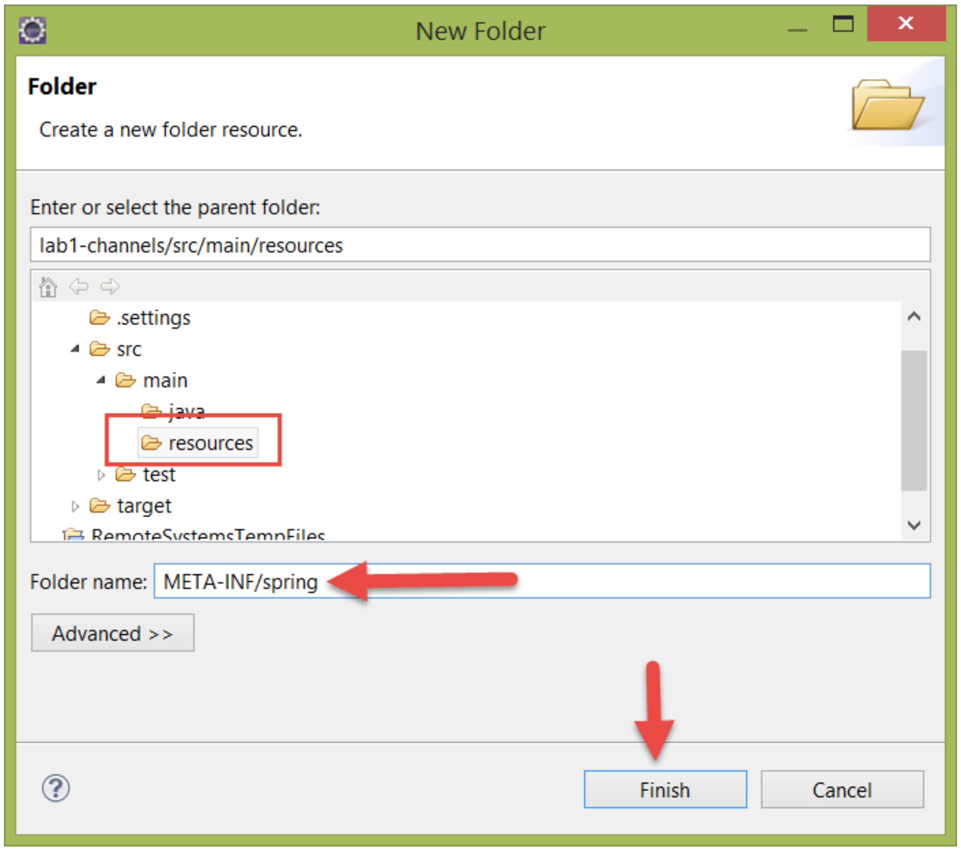


1.5 Create a META-INF/spring sub-folder in the resources directory. This folder will eventually contain your XML Spring configuration files.

1.5.1 Right click on the src/main/resources folder and select *New> Folder*.



1.5.2 In the New Folder window that displays, make sure the resources folder is still selected and then enter *META-INF/spring* as the new folder name and push the *Finish* button.



Your Maven project is now ready for Spring Integration development.

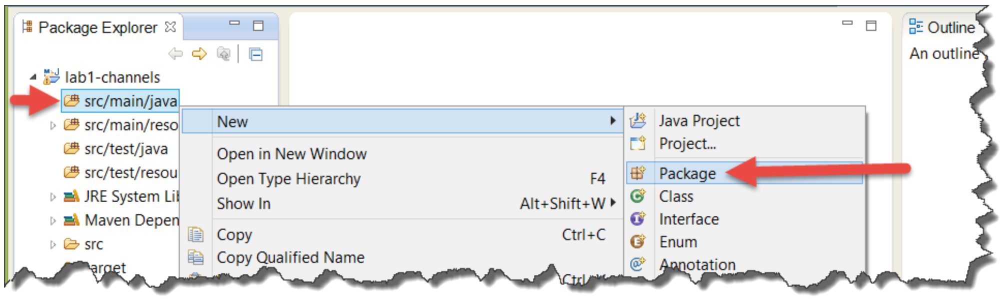
Step 2: Write the Java application.

In this step, you create a simple Java application that will create the Spring ApplicationContext and then put the JVM in an infinite loop. This will allow the Spring Integration components to produce, consume and react to messages that it sees while the JVM is up and running.

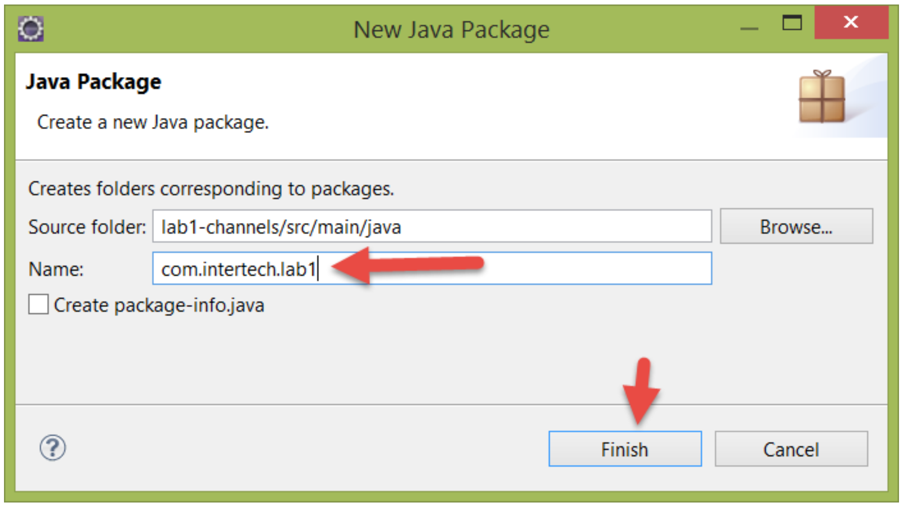
Note: If you do not wish to code this class by hand, you can find the code in Express Spring Integration\lab1\lab1-channels-helper-code

2.1 Create a new package in src/main/java.

2.1.1 Right click on the src/main/java folder in the Package Explorer and select *New > Package.*



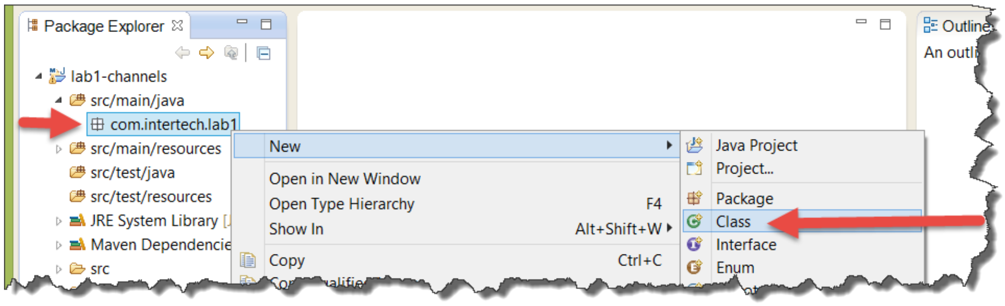
2.1.2 Enter *com.intertech.lab1* as the new package name and push the *Finish* button.



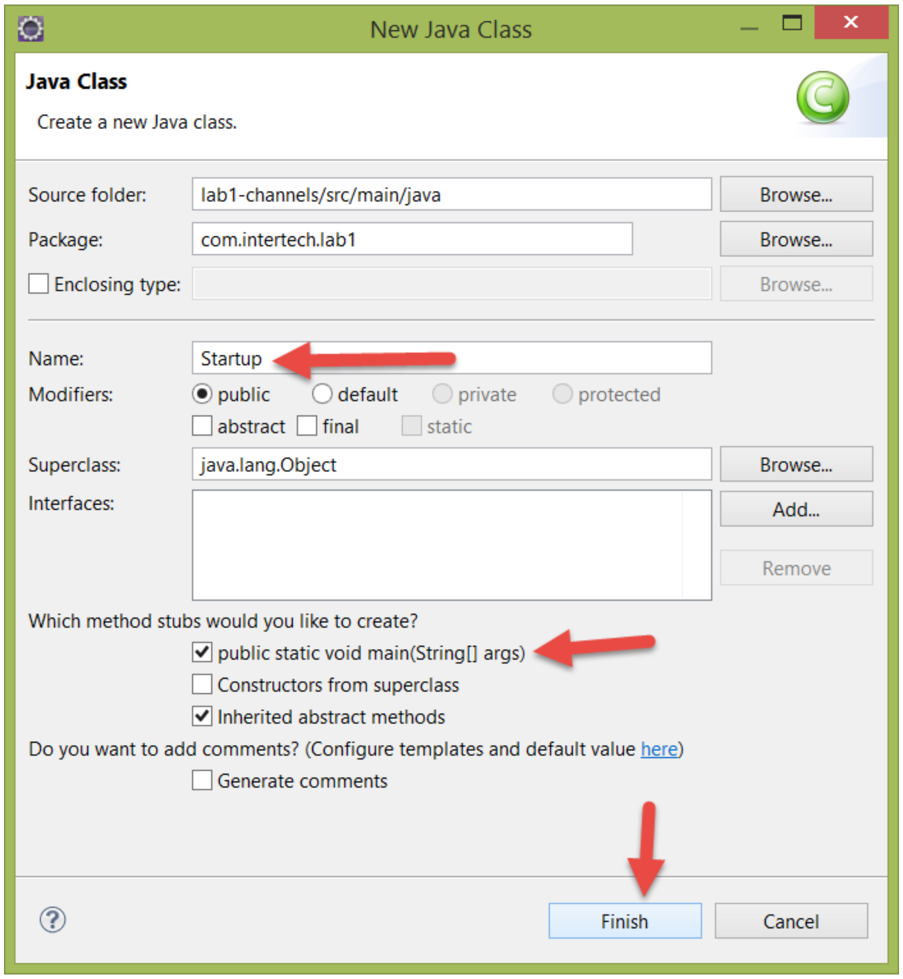
Creates folders corresponding to packages.
Source folder: lab1-channels/src/main/java
Name: com.intertech.lab1
☐ Create package-info.java

2.2 Create the Startup.java class. The Startup.java class will serve as a simple Java SE application – complete with a main( ) method. This class will be executed when the Spring Integration application is to run.

2.2.1 Right click on the newly created com.intertech.lab1 package and select New > Class from the resulting menu.



2.2.2 In the New Java Class window, enter *Startup* as the new class name, check the box to add a main( ) method and then push the *Finish* button.



Source folder: lab1-channels/src/main/java
Package: com.intertech.lab1
[ ] Enclosing type:
Name: Startup
Modifiers: [✓ public] default private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:
Which method stubs would you like to create?
[✓] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

2.3 Code the Startup.java class.

2.3.1 In the editor that opens, replace the TODO comment in the main method with the following code:

```
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("/META-INF/spring/si-components.xml");
while (true) {
}
```

2.3.2 Add the necessary imports to the class. Use Control-Shift-O to add the ClassPathXmlApplicationContext import to the top of the class.

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

2.3.3 Remove the compiler warning by adding an annotation to the main( ) method to suppress the warnings.

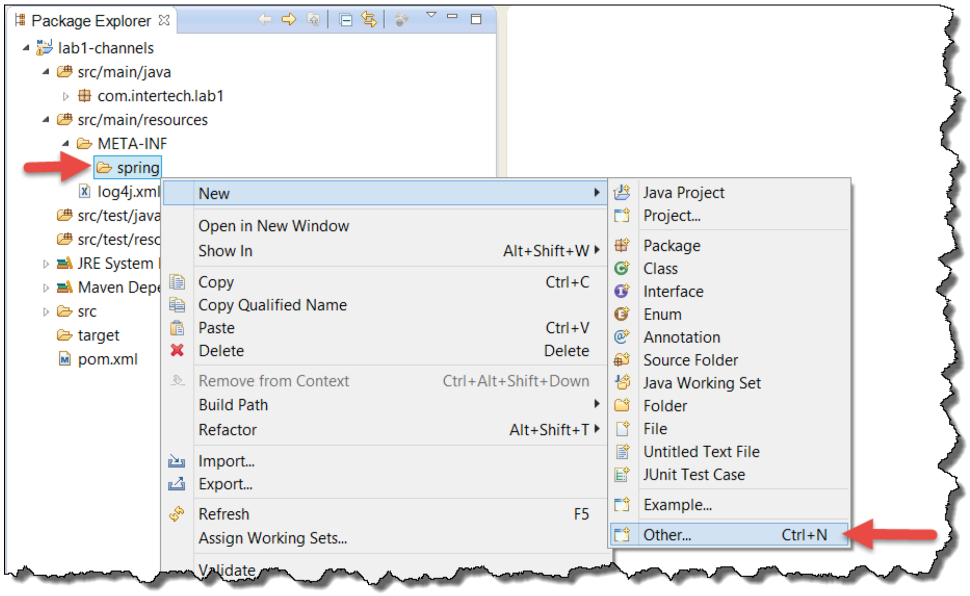
```
@SuppressWarnings({ "resource", "unused" })
public static void main(String[] args) {
  ...
}
```

2.3.4 Save Startup.java and make sure there are no compile errors in the project.

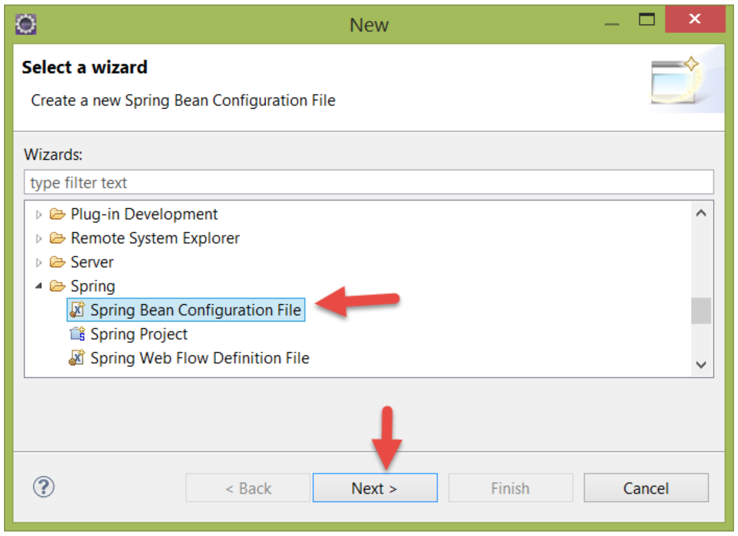
Step 3: Configure the Producer, Consumers, and Subscribable Channel

3.1 Create a Spring XML configuration file to hold all the SI components.

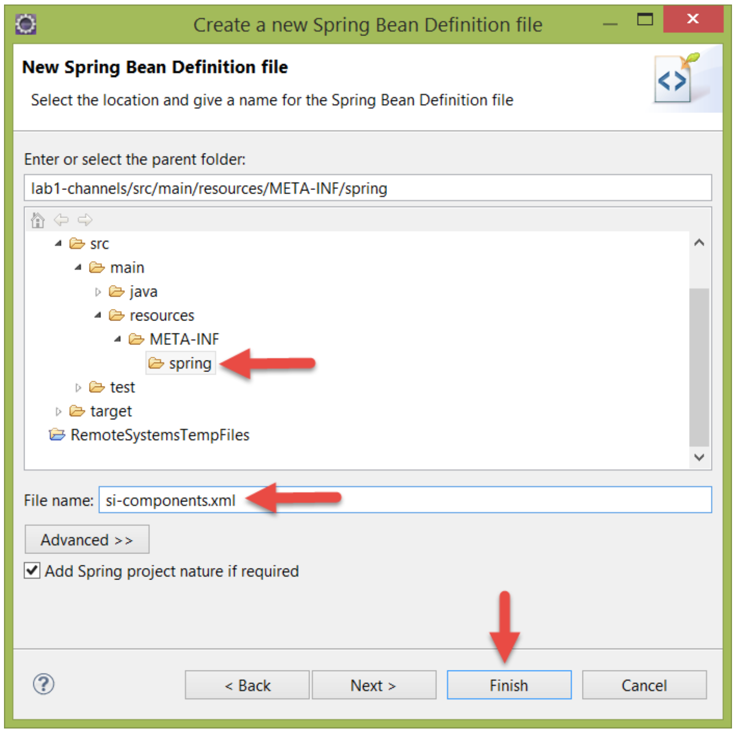
3.1.1 Right click on the spring folder located in src/main/resources/META-INF and select *Other…*



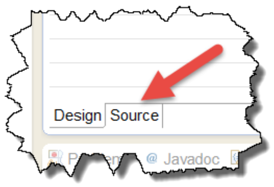
3.1.2 In the New window, expand the Spring folder, select *Spring Bean Configuration File* and then push the *Next>* button.



3.1.3 In the Create a new Spring Bean Definition file window, make sure the spring folder is selected, enter "si-components.xml" in the File name field and then push the *Finish* button.


Create a new Spring Bean Definition file
New Spring Bean Definition file
Select the location and give a name for the Spring Bean Definition file
Enter or select the parent folder:
lab1-channels/src/main/resources/META-INF/spring
src
main
java
resources
META-INF
spring
test
target
RemoteSystemsTempFiles
File name: si-components.xml
Advanced >>
Add Spring project nature if required
< Back
Next >
Finish
Cancel

3.1.4 In the XML editor that opens, select the Source tab to be able to edit the contents.

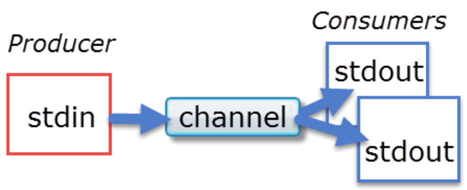


3.1.5 Enter the integration and stream namespaces to the <beans> root element as highlighted below.

Note: if you do not wish to enter the SI configuration elements, you can copy and paste them from a copy of the si-components.xml located in Express Spring Integration\lab1\lab1-channels-helper-code.

```
<beans
xmlns=http://www.springframework.org/schema/beans
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:int="http://www.springframework.org/schema/integration"
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">
```

3.2 Create a SI producer and consumer. In this lab, you will use the Java Standard Input stream and Standard Output stream as the producer and consumer of messages through channels. The Standard Input stream provides a means to enter text (through the Console view) that will become the message that enters the message channel. The text message will be consumed on the other end of the message channel and displayed to the Standard Output stream. An adapter provided by Spring Integration assist getting the message from Standard Input and enter it into the message channel. Another adapter provided by Spring Integration will get the message from the channel and display its contents to the Standard Output stream (the Console view). You will learn about the creation of adapters in a later lab. For now, just add the adapters that create/display text via the Standard Input/Output streams.



3.2.1 Inside of the <beans> element in the configuration file, enter a stdin-channel-adapter that wraps the Java Standard Input stream; taking text entered into the Standard Input stream and putting a text a message into the associated channel.

```
<int-stream:stdin-channel-adapter id="producer"
  channel="messageChannel" />
```

3.2.2 Add a poller that is used by the stdin-channel-adapter to determine how often to check the Standard Input stream for text (in this case, it checks every 200 milliseconds).

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

3.2.3 Create a couple of stdout-channel-adapters that wrap the Java Standard Output stream. They take the messages from the associated channel and display it to the Console.

```
<int-stream:stdout-channel-adapter
  id="consumer1" channel="messageChannel" append-newline="true" />
<int-stream:stdout-channel-adapter
  id="consumer2" channel="messageChannel" append-newline="true" />
```

3.3 Create the Subscribable Channel. With the producer and consumers in place, you are ready to create a channel to deliver the messages from producer to consumers. In this part of the lab, you create a subscribable channel. Subscribable channels do not buffer or hold onto messages. They simply deliver them to all the subscribers – that is the adapters on the consuming end of the channel. Subscribers deliver the messages to all subscribers. In this case, you have two subscribers – the two stdout-channel-adapters you created in the previous steps.

3.3.1 Add the publish-subscribe-channel element to the configuration below the adapters.

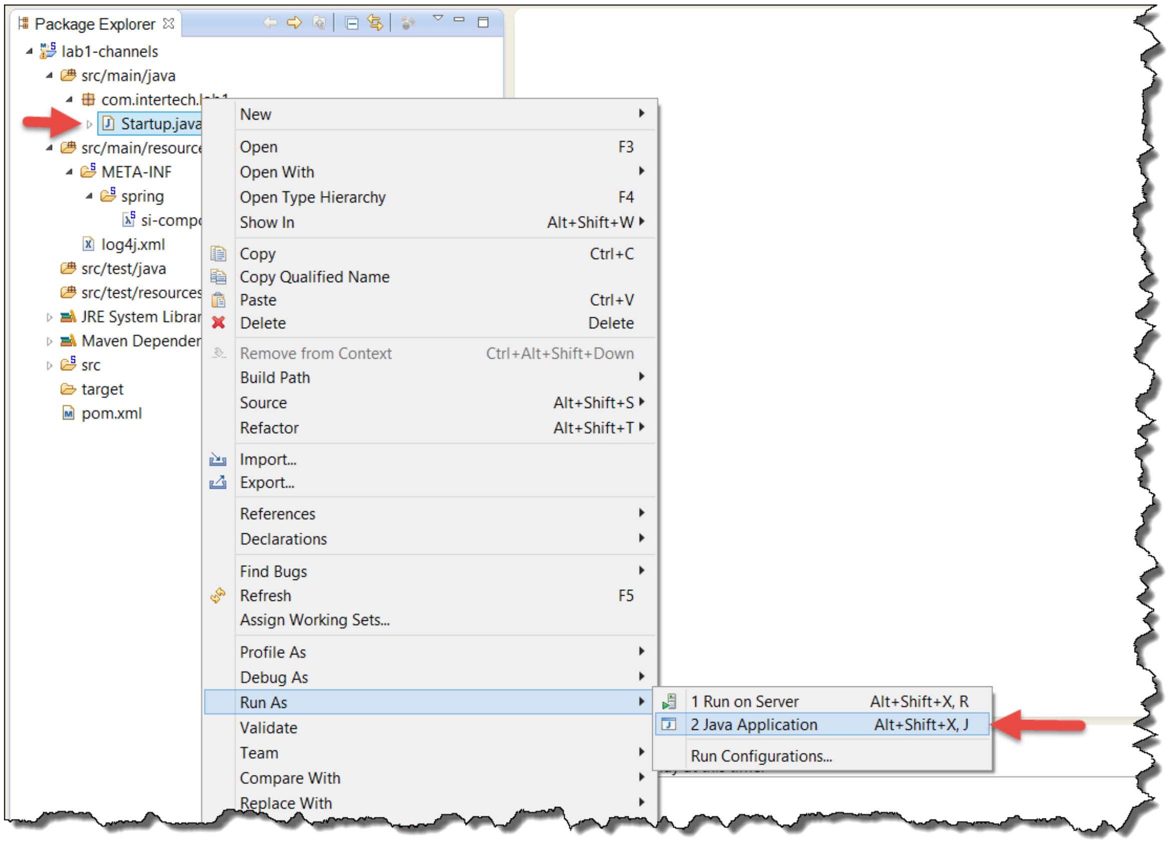
```
<int:publish-subscribe-channel id="messageChannel" />
```

Note that the id of the channel ("messageChannel") is the same as the channel referenced in both the stdin-channel-adapter and stdout-channel-adapters above.

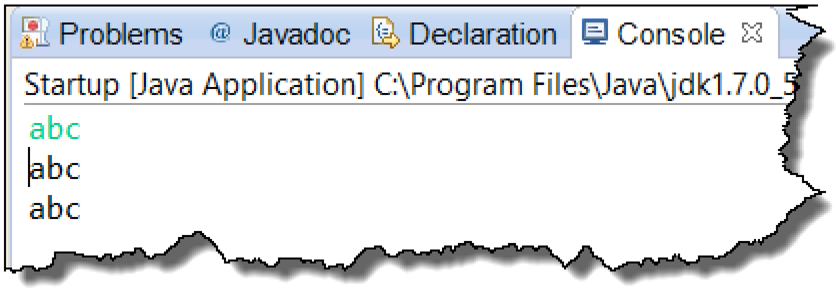
3.3.2 Save the configuration file and make sure there are no errors in the project.

3.4 Test the application. With the producer (stdin-adapter-channel), the consumers (stdout-adapter-channels), and the subscribable channel in place, it is time to test the application and your first SI channel.

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



3.4.2 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). A text message created from the text you enter into the Standard Input will immediately be marshalled into a text message and entered into the subscribable channel. Immediately after that, it will be delivered to the subscribers which display the output back to the Console view (in black). Since you have two subscribers, you should see the text you entered echoed back to the Console twice (as shown below). Feel free to enter additional text into the Standard Input and see more messages enter the channel.

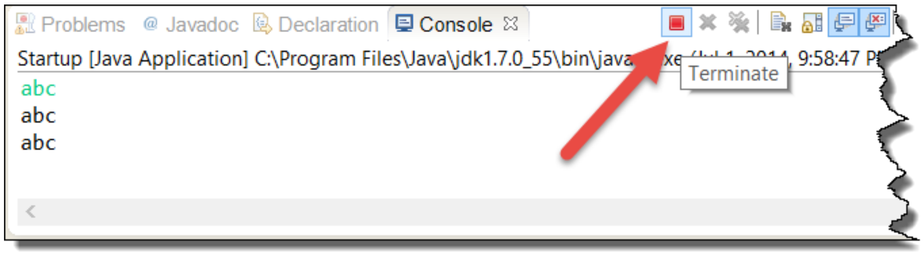


```
Startup [Java Application] C:\Program Files\Java\jdk1.7.0_5
abc
abc
abc
```

Congratulation! You just created your first Spring Integration application and used the first message channel.

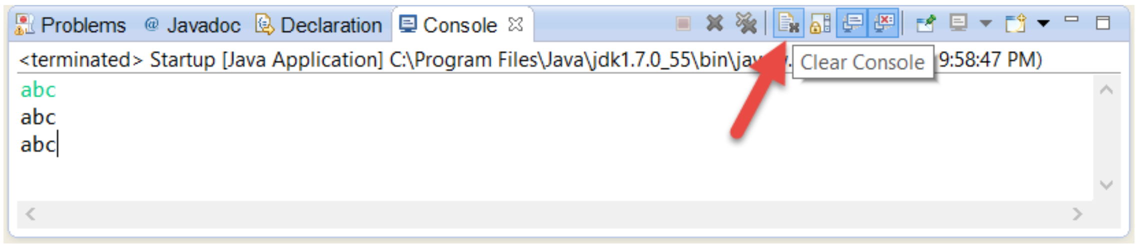
3.5 Stop the application. Recall the application is running in an infinite loop to allow for the constant publishing and consuming of messages. Stop the application now.

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



```
Startup [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\java  xe (Jul 1, 2014, 9:58:47 P
abc
abc
abc
```

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



```
<terminated> Startup [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\ja  9:58:47 PM)
abc
abc
abc
```

Step 4: Replace the Subscribable Channel with a Pollable Channel

Subscribable channels don't buffer messages and deliver the messages to any and all subscribers. Pollable channels, can buffer messages and deliver the message to a single subscriber. If there are more than one subscribers, it picks the first subscriber and skips the others. In this step, you replace the subscriber channel with a pollable channel to see the effect on the SI application.

4.1 Remove or comment out the Subscribable Channel.

4.1.1 Locate (in the src/main/resources folder) and open the si-components.xml in an editor by double clicking on the file.

4.1.2 Remove the subscribable channel by placing XML comments around it (<!-- -->).

```
<!--    <int:publish-subscribe-channel id="messageChannel" /> -->
```

4.2 Add a Pollable Channel. Pollable channels require a queue for potentially buffering messages with a configuration determined capacity.

4.2.1 Add the following elements to add the pollable channel with a buffering capacity of two (2) messages.

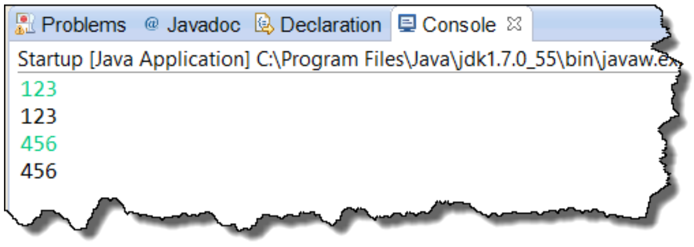
```
<int:channel id="messageChannel">
  <int:queue capacity="2" />
</int:channel>
```

4.2.2 Save the XML configuration file and make sure there are no errors in the project.

4.3 Retest the application. With the pollable channel in place of the subscribable channel, rerun the application.

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

4.3.2 The application is now running awaiting your text input. In the Console view, again enter some text and then hit the *Enter* key. A text message created from the text you enter into the Standard Input will immediately be marshalled into a text message and entered into the pollable **channel. This time, however, the message is only delivered to a single (the first) consumer of the messages from the channel. Note that the text is only echoed one time.**



```
Startup [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw.e
123
123
456
456
```

4.3.3 Terminate the application and clear Console view.

4.4 Remove the Producer and Consumer. In order to see the buffering of messages, it is necessary to change the producers and consumers. By default, they automatically publish and pull messages from the pollable channel without giving you a chance to see the messages stack up in the queue associated with the channel. In this step, you use the Statup.java file to publish messages onto the channel. Without a consumer, this will allow you to see messages stack up (and fill) in the pollable channel's queue.

4.4.1 Locate (in the src/main/resources folder) and open the si-components.xml in an editor by double clicking on the file.

4.4.2 Comment out the producer (stdin-channel-adapter) and consumer (stdout-channel-adapter) adapters.

```
<!-- <int-stream:stdin-channel-adapter id="producer" -->
<!--     channel="messageChannel" /> -->

<!-- <int-stream:stdout-channel-adapter -->
<!--     id="consumer1" channel="messageChannel" append-newline="true"
/> -->
<!-- <int-stream:stdout-channel-adapter -->
<!--     id="consumer2" channel="messageChannel" append-newline="true"
/> -->
```

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

4.5 Add code to Startup to produce messages into the Pollable Channel.

4.5.1 Locate and open the Startup.java file in an editor (in the src/main/java folder) by double clicking on the file.

4.5.2 Comment out the infinite loop.

```
//   while (true) {
//   }
```

4.5.3 Add the following code to programmatically add three messages into the pollable channel.

```
public static void main(String[] args) {
  ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
      "/META-INF/spring/si-components.xml");
  // while (true) {
  // }
  MessageChannel channel = context.getBean("messageChannel",
      MessageChannel.class);
  Message<String> message1 = MessageBuilder.withPayload(
      "Hello world - one!").build();
  Message<String> message2 = MessageBuilder.withPayload(
      "Hello world - two!").build();
  Message<String> message3 = MessageBuilder.withPayload(
      "Hello world - three!").build();
  System.out.println("sending message1");
  channel.send(message1);
  System.out.println("sending message2");
  channel.send(message2);
  System.out.println("sending message3");
  channel.send(message3);
  System.out.println("done sending messages");
}
```

4.5.4 Add the necessary imports (all imports required are shown below) by hitting Control-Shift-O.

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageChannel;
```

4.5.5 Optionally, you can also remove the “unused” portion of the @SuppressWarnings annotation.

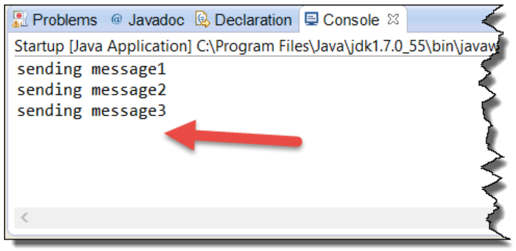
@SuppressWarnings({ "resource"~~, "unused"~~ })

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

4.6 Retest the application (and see the queue limit being reached).

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

4.6.2 This time, the application does not require your input (the messages are added to the pollable channel by the Startup code). Look in the Console view to see what is happening.



```
Startup [Java Application] C:\Program Files\Java\jdk1.7.0_55\bin\javaw
sending message1
sending message2
sending message3
```

Notice that the System outputs show the sending of messages, but not the last output before the application ends (the "done sending messages" text). Why? The queue is only two deep. The application is waiting for a consumer to empty a message out of the channel so the last message can be placed into the channel. The pollable channels are buffered, but only as big as you set their capacity.

4.6.3 Terminate the application and clear Console view.

Step 5: Replace the Pollable Channel with a Direct Channel

Direct channels are the default channel type in SI. It is a subscribable channel but it acts more like a point-to-point channel (like a pollable channel). In other words, it will send its message to a single subscriber (like a pollable channel). In this step, you replace the pollable channel with a direct channel.

5.1 Return the Producer and Consumer adapters. Return the Standard Input and Output adapters to the configuration file. The adapters will again be used to push a message into the channel and remove the message from the channel.

5.1.1 Locate (in the src/main/resources folder) and open the si-components.xml in an editor by double clicking on the file.

5.1.2 Uncomment the producer (stdin-channel-adapter) and consumer (stdout-channel-adapter) adapters.

```
<int-stream:stdin-channel-adapter id="producer"
  channel="messageChannel" />

<int-stream:stdout-channel-adapter
  id="consumer1" channel="messageChannel" append-newline="true" />
<int-stream:stdout-channel-adapter
  id="consumer2" channel="messageChannel" append-newline="true" />
```

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

5.2 Remove the message producing code in Startup. Return the Startup.java class to its original code.

5.2.1 Remove the message producing code and uncomment the infinite while loop in the main( ) method. The main( ) method should look like that below when finished.

```
public static void main(String[] args) {
  ClassPathXmlApplicationContext context = new
    ClassPathXmlApplicationContext("/META-INF/spring/si-
components.xml");
  while (true) {
  }
}
```

5.2.2 Remove any unnecessary imports from the class. The following is the only import required.

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

5.2.3 You may optionally want to return the @SuppressWarnings annotation to its original form as well.

```
@SuppressWarnings({ "resource", "unused" })
```

5.2.4 Save the file and make sure there are no compile errors.

5.3 Replace the Pollable Channel with a Direct Channel.

5.3.1 In the si-components.xml file, remove the queue from the pollable channel (the "messageChannel") in order to create a direct channel. Direct channels are the default implementation for <int:channel> when using SI XML configuration.

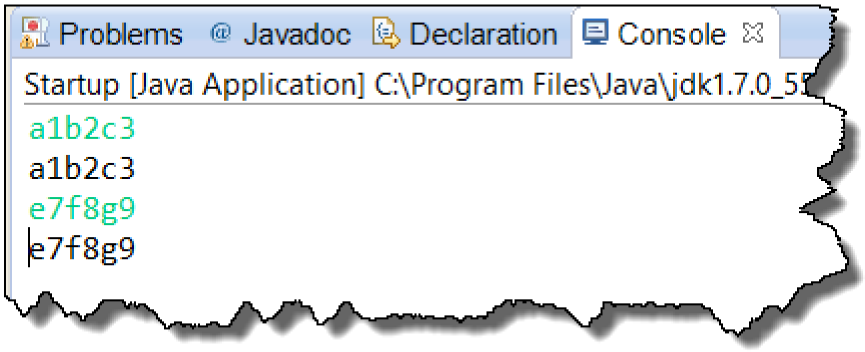
```
<int:channel id="messageChannel">
  <!-- <int:queue capacity="2" /> -->
</int:channel>
```

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

5.4 Retest the application. One last time, run Startup.java as a Java application.

5.4.1 Right click on file and select Run As > Java Application from the resulting menu.

5.4.2 The application is now running awaiting your text input. In the Console view, again enter some text and then hit the *Enter* key. A text message created from the text you enter into the Standard Input will immediately be marshalled into the direct channel. Notice how the message is only delivered to a single (the first) consumer of the messages (like a pollable channel but without a queue!). Note that the text is only echoed one time.

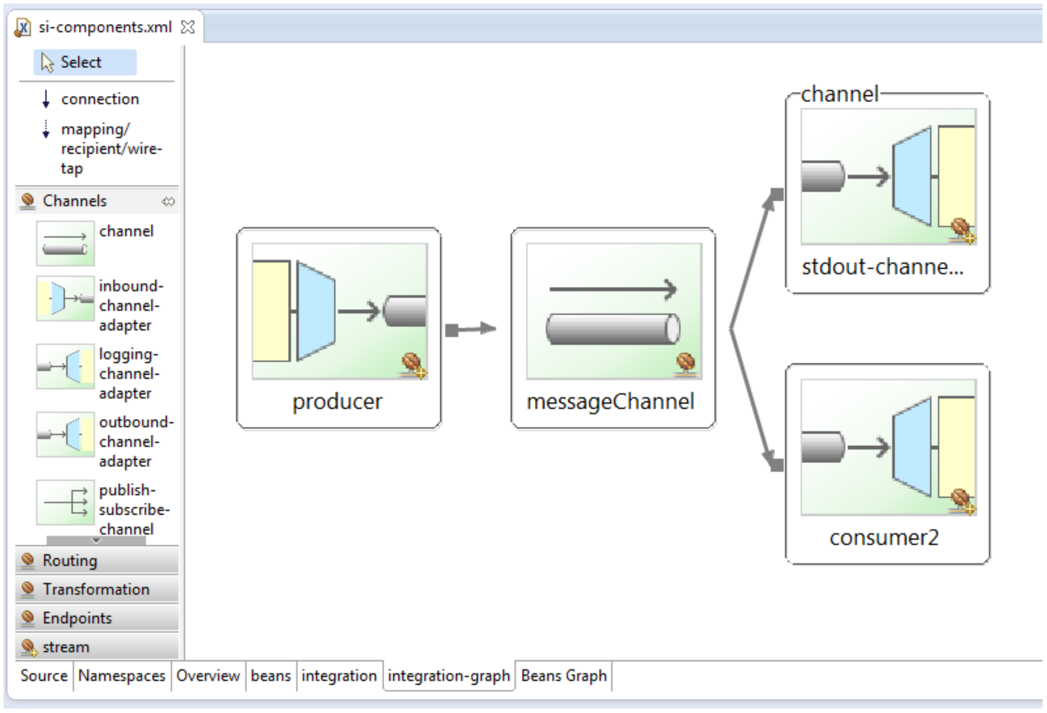


```
Startup [Java Application] C:\Program Files\Java\jdk1.7.0_55
a1b2c3
a1b2c3
e7f8g9
e7f8g9
```

5.4.3 Terminate the application and clear Console view.

5.5 Recall that SI implements the Hohpe/Woolf Enterprise Integration Patterns (EIP). Therefore, when designing or documenting a Spring Integration application, we represent the SI components in a model diagram using Hohpe Enterprise Integration Pattern (EIP) icons. There are several tools that can be used to create the diagrams.

- You can download a Hohpe/Woolf Visio stencil set that can be used with Microsoft Visio to create the diagrams (http://www.eaipatterns.com/downloads.html).
- If you are using Spring Tool Suite (STS), the XML configuration file will automatically be displayed in an EIP diagram as you assemble your components in XML. If using STS, open the configuration file and then click on the integration-graph tab. Below is the EIP model diagram for the application created in this lab.



Channels make up a significant part of any SI project. They provide the pipes between all the working components of an EAI project. Transformers, filters, enrichers, etc. may do the business work of a SI application, but it is channels that get information and process requests to these components. You have now seen three different type of channels (publish-subscribe, pollable, and direct channels) in SI and seen some of the differences in how they behave and operate. There are other types of channels in SI, but they all descend from either subscribable or pollable channels which you have now learned.

Lab Solution

Startup.java (in its final form)

```
package com.intertech.lab1;

import
org.springframework.context.support.ClassPathXmlApplicationContext;
//import org.springframework.integration.support.MessageBuilder;
//import org.springframework.messaging.Message;
//import org.springframework.messaging.MessageChannel;

public class Startup {

  @SuppressWarnings({ "resource", "unused" })
  // @SuppressWarnings({ "resource" })
  public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
        "/META-INF/spring/si-components.xml");
    while (true) {
    }
    // MessageChannel channel = context.getBean("messageChannel",
    // MessageChannel.class);
    // Message<String> message1 = MessageBuilder.withPayload(
    // "Hello world - one!").build();
    // Message<String> message2 = MessageBuilder.withPayload(
    // "Hello world - two!").build();
    // Message<String> message3 = MessageBuilder.withPayload(
    // "Hello world - three!").build();
    // System.out.println("sending message1");
    // channel.send(message1);
    // System.out.println("sending message2");
    // channel.send(message2);
    // System.out.println("sending message3");
    // channel.send(message3);
    // System.out.println("done sending messages");
  }
}
```

si-components.xml (in final form)

```
<?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"
  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">

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

  <!-- a pair of message consumers / a pair of Spring Integration
wrapped Java Standard output streams -->
  <int-stream:stdout-channel-adapter
    id="consumer1" channel="messageChannel" append-newline="true" />
  <int-stream:stdout-channel-adapter
    id="consumer2" channel="messageChannel" append-newline="true" />

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

  <!-- a pub/sub message channel -->
  <!-- <int:publish-subscribe-channel id="messageChannel" /> -->

  <!-- a direct channel without the queue, a pollable channel with the
queue -->
  <int:channel id="messageChannel">
    <!-- <int:queue capacity="2" /> -->
  </int:channel>

</beans>
```