Spring Core Unit – Software Development Lifecycle

Lesson 3: Dependency Injection

The Learning House
427 S 4th Street #300
Louisville KY 40202

Lesson 3: Dependency Injection

Overview

One of the core design concepts that we have discussed and used so far has been tiered application design. We have said that it is important to maintain loose coupling between layers so the implementation of each layer can change without breaking the application. The design of encapsulated classes, which have public interfaces and private implementations, is key to achieving this. Further, programming to Java interfaces, rather than concrete classes, gives us a language mechanism to achieve this.

Although this approach adds great flexibility to our programs, our objects are still responsible for directly instantiating the other objects on which they depend, which still couples them to a particular concrete implementation. By using dependency injection, objects are no longer responsible for creating the other objects on which they depend — the DI container (Spring) hands the required references to the objects instead. Creating these relationships between application objects is the heart of dependency injection and is commonly referred to as bean wiring.

In this lesson, we'll look at some simple examples of wiring beans for an application. This is a foundational Spring skill — almost all other Spring features (security, AOP, MVC, REST) take advantage of the core Spring DI container, which means that this is something you'll do in just about every Spring project you encounter.

Olympian Example

To illustrate how the Spring DI container works, we will write a simple Console application that uses the container. We'll also take a look at how we can use JUnit to test applications built in this manner. The actual code in this example application is quite simple — the focus of this example is on the organization and configuration of the project. You should use this as a guide for configuring the Spring DI container for future projects. This example will also be used to demonstrate Spring aspect-oriented programming (AOP) features in the next lesson.

The purpose of our program is to create a simple model of Olympic events and Olympic athletes. Our goal will be to make the athlete objects (Olympians) as flexible as possible. We'll start with a rather brittle (i.e. hard-coded) implementation and then use Spring DI to make it more flexible as we go.

Project Setup

The first step in this process is the creation of a new Maven project in NetBeans. Simply follow these steps (a more detailed explanation can be found in **Spring Core Unit - Lesson 02: Intro to Maven**):

1. Select File → New Project
2. In the New Project dialog, select Maven under Categories and Java Application under Projects. Click Next.
3. In the New Java Application dialog, enter Olympian for the Project Name, choose a location for the project, and enter the Group Id (e.g. com.swcguild) and version (e.g. 1.0) of your choice. Click Finish.

The NetBeans 8.0 project creation wizard for Maven Java Applications does not include any dependencies in the POM file, which means we have to add them manually. At this point, we have to add a dependency for JUnit. Please add the following XML snippet to your pom.xml file (insert it after the <properties> node, just before the closing <project> tag):

Notes:

1. You must add the <dependencies></dependencies> container tag. This contains all dependency entries for the project.
2. Each Maven dependency entry consists of three sub-nodes:
    a. **groupId**: general identifier for the library provider or project
    b. **artifactId**: identifier for this library
    c. **version**: version of the library you want to use
3. Maven dependency entries can optionally have a **scope** node, which indicates which part of the lifecycle the dependency should be associated with. In our example, JUnit is only needed in the test phase of the lifecycle. If the dependency is needed in all phases, no **scope** node is necessary.

```
<dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
</dependencies>
```

Side Note: How do I know what my dependency entry should be?

All of the libraries that will be used in this class are available in online Maven repositories. This will be true of the vast majority of other open-source libraries as well. The question is — how do I know what the Maven dependency entry is supposed to be? There are two good approaches for finding this out:

1. Go to the project web site — many of them will have the dependency entry there. Simply copy and paste into your POM file. For example, the Maven entry for the Spring core framework is found here: http://projects.spring.io/spring-framework/
2. Simply search for the dependency on Google. For example, a search for junit 4.11 maven dependency yields this page as its top result: http://mvnrepository.com/artifact/junit/junit/4.11 which contains the Maven dependency entry for JUnit 4.11.

Initial Implementation Without Spring

To start, we want to model an **event** and an **athlete** to participate in that event. Since there are potentially many events in any given Olympics, we'll start with an event interface and then create a concrete implementation. You will see that this particular approach causes our athlete (SkiJumper) to be tightly coupled to the SkiJump event.

Event Interface

Notes:

1. All events in our program will implement this interface.
2. We will always program to this interface rather than to a particular concrete implementation of the interface.
3. The interface has only one method: compete. The String value returned should be the name of the event (we'll use this for testing).

```
public interface Event {

    public String compete();

}
```

SkiJumpEvent Implementation

Notes:

1. We print out a message to the console in the **compete** method so that we can see when the athlete actually competes in the event when the program is run.

```
public class SkiJumpEvent implements Event {
    public String compete() {
        System.out.println("SkiJuuuummmmmppppiiiinnnnnggggg!!!!!");
        return "SkiJump";
    }
}
```

SkiJumper Implementation

We now need an athlete to compete in the Ski Jump Event, so we'll create a SkiJumper object:

Notes:

1. Even though SkiJumper is using the Event interface reference for its Event, it is still completely responsible for instantiating the correct Event implementation.

```
public class SkiJumper {

    private Event event;

    public SkiJumper() {
        event = new SkiJumpEvent();
    }

    public String competeInEvent() {
        return event.compete();
    }
}
```

Unit Test

There isn't much to test with our SkiJumper at the moment, but we'll implement a unit test anyway. This will become more useful in later steps.

```
@Test
public void SkiJumperTest() {
    SkiJumper jumper = new SkiJumper();
    assertEquals(jumper.competeInEvent(), "SkiJump");
}
```

SkiJumper Implementation With Spring Container

In the second version of our application, we will let the Spring container control the lifecycle of the SkiJumper object. This step will not take advantage of the DI features of the container, but it puts all the needed Spring dependencies and configuration files in place so we can use DI in the third version of the application. The only difference between this version and the first is that our code will not explicitly instantiate the SkiJumper; instead, the Spring container will instantiate the object and our code will just ask for a reference.

Step One: Add Spring Dependencies Entries to POM

In order to use the Spring container, we must add the Spring dependency to our Maven POM file. Please add the following entry to your pom.xml file (make sure this entry is inside the <dependencies> tag):

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.0.3.RELEASE</version>
</dependency>
```

Step Two: Add Application Context File to Source Packages

To use the Spring container in our application, we must include a Spring configuration file. The Spring container is generally referred to as the Application Context and we will call the configuration file applicationContext.xml (although you can call it anything you want). To start, we'll just add the skeleton of the file to our project. Please create a file called applicationContext.xml in the **src** → **main** → **resources** folder of your project (you may have to create the folder) and copy the following content into it:

Notes:

1. This is an empty template for the applicationContext file which is prepopulated with Spring XML namespace entries for the Spring container as well as some additional Spring features we'll use later in the class. Use this as a starting point for all of your Spring configuration files.
2. All bean wiring definitions will appear after the "Bean definitions go here" comment and before the closing </beans> tag

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd
  http://www.springframework.org/schema/aop

  http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">

    <!-- Bean definitions go here -->

</beans>
```

Step Three: Add Application Context File to Test Packages

Maven projects make a distinction between the environment used for running unit tests and the environment used for running the application itself. This means that we need a separate Spring configuration file for the unit test environment. Please create a file called applicationContext.xml in the **src** → **test** → **resources** folder of your project (you may have to create this folder). Once this is created, copy in the contents from Step Two, above. The two applicationContext.xml files should be identical.

Step Four: Wire SkiJumper Bean

Now that we have our configuration files in place, we need to tell the Spring container that we would like it to instantiate our SkiJumper object for us. The process is referred to as **wiring**. The minimum entry for any bean in the Spring application context file consists of an **id** — which is simply an alias that we can use to refer to the instance — and the **fully-qualified name of the class** that defines the object. The entry for our SkiJumper looks like this (change the fully-qualified class name to match your project):

```
<bean id="superSkiJumper" class="com.swcguild.olympian.SkiJumper"/>
```

Add the above entry to both of your applicationContext.xml files (insert just after the “Bean definitions go here” comment).

Step Five: Implement main()

Now, we will add code to the main() method of our application that will instantiate the Spring container and then ask for a reference to our superSkiJumper object. In order to do this, we create a new ClassPathXmlApplicationContext object and pass it the name of our Spring configuration file. The resources folders in which we placed our application context files are both on the **classpath** of our application. This means that Spring can load the configuration files from the classpath of the application rather than directly from the file system.

Add the following code to your main() methods and run your application.
“SkiJuuuummmmmpppppiiiinnnnnggggg!!!!!” should be printed to the console.

Notes:

Always use the ApplicationContext interface reference rather than a reference to a concrete implementation.

ClassPathXmlApplicationContext will look for the given file name on the application classpath.

The getBean method returns an Object reference, which means that we have to cast it to the correct type (in this case SkiJumper) in order to use it.

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
SkiJumper sj = (SkiJumper) ctx.getBean("superSkiJumper");
sj.competeInEvent();
```

Step Six: Implement Test

Our test implementation will look very similar to the code in the main() — we just add a call to assertEquals to test the result of the competeInEvent() method:

```
@Test
public void SkiJumperTest() {
    ApplicationContext ctx = new

                ClassPathXmlApplicationContext("applicationContext.xml");
    SkiJumper sj = (SkiJumper) ctx.getBean("superSkiJumper");
    assertEquals(sj.competeInEvent(), "SkiJump");
    }
```

Implement with Dependency Injection

Although we are now using the Spring application context to manage the lifecycle of our SkiJumper, the implementation is still tightly coupled to the SkiJumpEvent — in other words, our SkiJumper object is still fully responsible for instantiating the SkiJumpEvent even though it is accessing the SkiJumpEvent through an Event reference.

In this step, we will write code to make our application more flexible and make our athlete less tightly coupled to a particular event.

The Olympian Class

Our first step is to create a more flexible model for our athlete. Rather than creating a new class for each type of athlete, we'll create one Olympian class. Further, rather than having the athlete's event hardcoded (as it was in the SkiJumper class), we'll pass the Event into the Olympian's constructor — this means that our athlete class is no longer responsible for instantiating its event because we'll hand it an event at construction time. Create a new Java class called Olympian and paste in the following code:

Notes:

1. Our class has a private class level reference to an Event, which gets initialized in the constructor.
2. Our class has a private class level variable for country — it defaults to USA but can be changed via a setter method.
3. The code that instantiates this class is responsible for handing us a reference to an Event of some sort — we are not responsible for instantiating our event.

```
public class Olympian {
    private Event event;
    private String country = "USA";

    public Olympian(Event event) {
        this.event = event;
    }

    public String competeInEvent() {
        System.out.println("Now competing for " + country + ":");
        return event.compete();
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

Instantiate and Use Olympian from Main

Now, we'll add code in our main method that instantiates and uses the Olympian. We'll create an Olympian that participates in the SkiJumpEvent to illustrate how this approach differs from our initial implementation. Note that the following code does not use the Spring application context at all — we are instantiating Olympian by hand. Also note that we can now create an Olympian that competes in any event without having to write additional athlete classes. Add the following code to main:

Notes:

1. We must first instantiate an object that implements the Event interface (in this case, the SkiJumpEvent).
2. We must pass a reference to our newly instantiated Event object into the constructor of our Olympian object.

```
public main() {
    Event skiJumpEvent = new SkiJumpEvent();
    Olympian olympianSkiJumper = new Olympian(skiJumpEvent);
    olympianSkiJumper.competeInEvent();
}
```

Create More Events

In order to fully take advantage of the flexibility of our Olympian class, we'll need a few more events. Create the following classes in your project:

```
public class SpeedSkateEvent implements Event {
    public String compete() {
        System.out.println("Skating REALLY fast!!!!");
        return "SpeedSkate";
    }
}

public class CrossCountrySkiEvent implements Event {
    public String compete() {
        System.out.println("Skiing.  Really Fast.  Cross Country.  Even up hills.");
        return "CrossCountrySki";
    }
}
```

Wire Event and Olympian Beans

Our goal is to have Spring manage the lifecycle of our Olympian objects. Now that these objects are no longer responsible for creating the Events in which they participate, we must have Spring manage the lifecycle of each Event object that we want to use in our program. To do this, we must add a bean entry for each Event. Add the following bean definitions to your applicationContext.xml file:

```
<bean id="speedSkating" class = "com.swcguild.olympian.SpeedSkateEvent"/>
<bean id="skiJumping" class = "com.swcguild.olympian.SkiJumpEvent"/>
<bean id="crossCountry" class = "com.swcguild.olympian.CrossCountrySkiEvent"/>
```

Finally, we'll wire some Olympian beans with different characteristics:

1. A ski jumper from the USA
2. A speed skater from the USA
3. A speed skater from Canada

For the first two bean definitions, we'll use **constructor injection** to pass in a reference to the correct Event (the references will be to the Event beans we wired in above). For the third bean definition, we'll use constructor injection for the Event and **setter injection** to change the country from USA to Canada. Add the following bean definitions to your applicationContext.xml file:

Notes:

1. The **constructor-arg** tag contains a "ref" attribute — the ref attribute is a reference to another bean that has been defined in the applicationContext file. In this case, the refs point to the Event beans we defined earlier.
2. The **property** tag (on the canadianSpeedSkater bean) contains a "value" attribute. In this case, the String "Canada" will be passed to the setCountry(...) method on the Olympian object created from this definition.

```
<bean id="usaSkiJumper" class="com.swcguild.olympian.Olympian">
    <constructor-arg ref="skiJumping"/>
</bean>



<bean id="usaSpeedSkater" class="com.swcguild.olympian.Olympian">
    <constructor-arg ref="speedSkating"/>
</bean>

<bean id="canadianSpeedSkater" class="com.swcguild.olympian.Olympian">
    <constructor-arg ref="speedSkating"/>
    <property name="country" value="Canada"/>
</bean>
```

Get References to Olympian Objects from Spring Application Context

Now we can get the references to our Olympian objects from the Spring container. We are not responsible for instantiating these objects, nor are we responsible for managing the Event objects on which the Olympian objects depend. The Olympian object are not responsible for the objects on which they depend either — we've externalized all of this to the applicationContext.xml file and we're letting Spring manage all of this for us.

To demonstrate this, add the following code to your main() method:

```
Olympian usaSkiJumper = (Olympian) ctx.getBean("usaSkiJumper");
usaSkiJumper.competeInEvent();

Olympian usaSpeedSkater = (Olympian) ctx.getBean("usaSpeedSkater");
usaSpeedSkater.competeInEvent();

Olympian canadaSpeedSkater = (Olympian) ctx.getBean("canadianSpeedSkater");
canadaSpeedSkater.competeInEvent();
```

Error Conditions

The Spring container is a powerful tool that helps us build flexible, loosely coupled applications. This is a great thing, but it does add more moving parts to our environment. One consequence of externalizing dependencies to an XML configuration file is that any errors contained in the configuration file are not detectable until runtime, because the compiler has no way of catching these errors.

Here are some common errors that you might encounter:

1. Misspelled bean class attribute — all bean classes must be specified by their correctly spelled, fully-qualified class name. If you misspell this, the Spring container will throw an error indicating that the bean cannot be instantiated.
2. Misspelled bean id in ref attribute — when you refer to another bean (in a ref attribute), it must appear exactly as it did in the original bean definition.
3. Malformed xml — the applicationContext.xml file must be composed of well-formed XML. These are the easiest errors to catch and diagnose — the error message usually tells you the line and column where the error occurred.
4. Using the **value** attribute when you meant to use the **ref** attribute in constructor or setter injection. Remember that the value attribute passes the literal value of the attribute to the constructor or setter whereas the ref passes a reference to the object referred to by the attribute value.

These types of error message will show up in the NetBeans console output.

Wrap-up

In this lesson, we looked at dependency injection and how we can use the Spring Container's DI features to externalize object dependencies and create loosely coupled applications. One of the main goals of this lesson was to understand how the Spring Container is configured. We created a simple project that can be used as a template for future Maven/Spring projects. Here are the important points:

1. From now on, all projects will be Maven projects.

2. We have to add dependency entries for all external libraries to the Maven POM file. Specifically for this project, we added entries for JUnit and Spring Core.
3. The Spring Container (known as the Application Context) is configured via an XML file that contains definitions for all of the objects we want Spring to control.
4. Maven makes a distinction between the application and test environments. We must have a Spring configuration file for each environment.
5. We place the Spring configuration files on the classpath. In our Maven projects, we place these files in either **Other Sources** or **Other Test Sources**.
6. To use the Spring Container, we must instantiate an implementation of ApplicationContext (in our case, we use ClassPathXmlApplicationContext). The ApplicationContext will read the Spring configuration file and instantiate all configured objects. We then ask the ApplicationContext for references to these objects by name rather than instantiating them ourselves.

In the next lesson, we'll explore aspect-oriented programming and extend the Olympian example to illustrate these features.