

Dependency Injection

Kenneth M. Anderson
University of Colorado, Boulder
Lecture 30 — CSCI 4448/5448 — 12/08/11

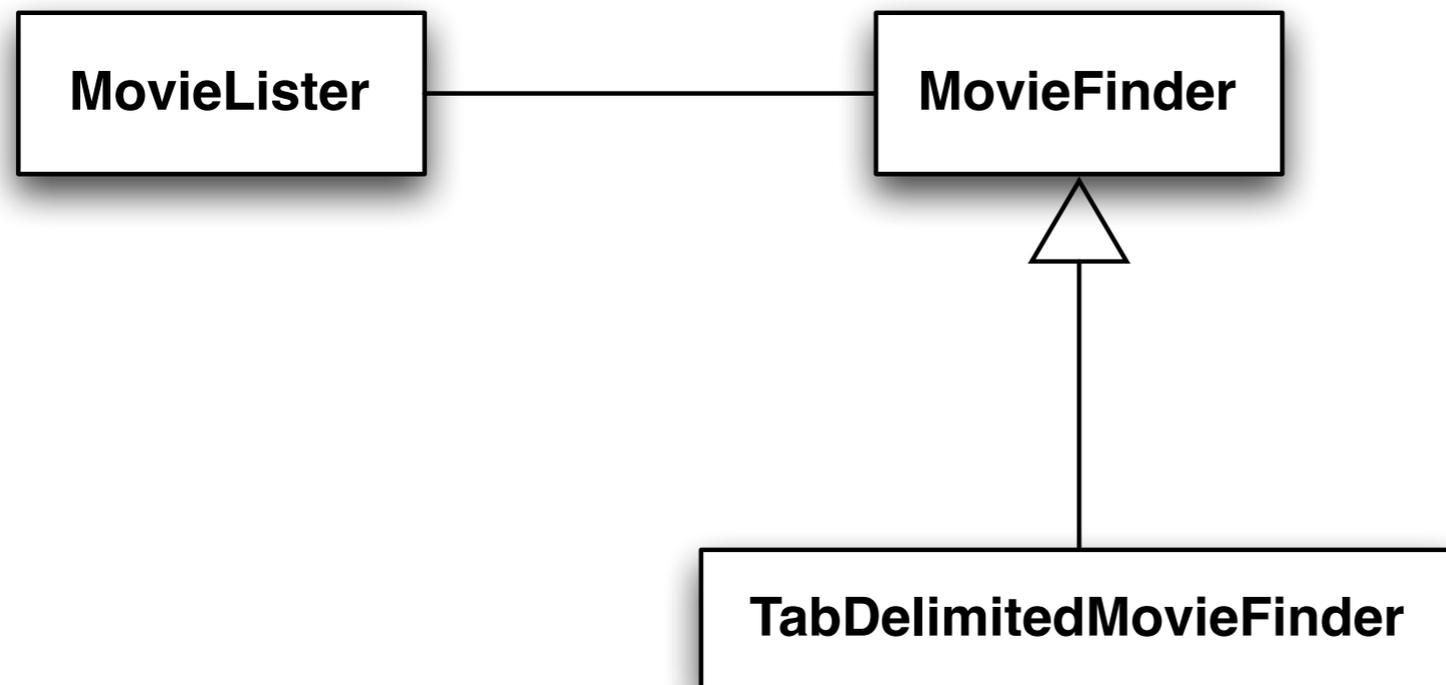
Goals of the Lecture

- Introduce the topic of dependency injection
- See examples using the Spring Framework
 - Note: I'm using Spring 2.5.6 for this lecture
 - The latest production release is 3.0.6
 - The latest development release is 3.1.0
 - I'm only going to scratch the surface of Spring's capabilities
 - It is an extremely powerful framework that provides TONS of functionality (more than just dependency injection)
- Note: you need to download the "with-dependencies" .zip file in order to acquire all of the .jar files you need to run the examples

Dependency Injection

- Dependency Injection is
 - a technique for assembling applications
 - from a set of concrete classes
 - that implement generic interfaces
 - without the concrete classes knowing about each other
 - This allows you to create loosely coupled systems as the code you write only ever references the generic interfaces that hide the concrete classes
 - Dependency Injection is discussed in a famous blog post by Martin Fowler
 - <http://martinfowler.com/articles/injection.html>

Fowler's Example



A **MovieLister** class is able to list movies with certain characteristics after being provided a database of movies by an instance of **MovieFinder**

MovieFinder is an interface; **TabDelimitedMovieFinder** is a concrete class that can read in a movie database that is stored in a tab-delimited text file

The Goal: Loosely-Coupled Systems

- Our goal (even with the simple system on the previous slide) is to avoid having our code depend on concrete classes

- In other words, we do **NOT** want to see something like this

- public class MovieLister {

- private MovieFinder finder;

- public MovieLister() {

- this.finder = new TabDelimitedMovieFinder("movies.txt");

- }

- ...

- }

Dependency on hard-coded string

Dependency on Concrete Class

Discussion

- The code on the previous slide has two concrete dependencies
 - a reference to a concrete class that implements MovieFinder
 - a reference to a hard-coded string
- Both references are brittle
 - The name of the movie database cannot change without causing MovieLister to be changed and recompiled
 - The format of the database cannot change without causing MovieLister to be changed to reference the name of the new concrete MovieFinder implementation

Our Target (I)

- For loose-coupling to be achieved, we need code that looks like this
- ```
public class MovieLister {
 • private MovieFinder finder;
 • public MovieLister(MoveFinder finder) {
 • this.finder = finder;
 • }
 • ...
• }
```
- and, furthermore, nowhere in our source code should the strings “TabDelimitedMovieFinder” or “movies.txt” appear... nowhere!

# Our Target (II)

---

- As much as possible, get rid of code with the form
    - `Foo f = new ConcreteFoo();`
  - Indeed, for the MovieLister system, we would even like to see code like this
    - `public class Main {`
      - `private MovieLister lister;`
      - `public void setMovieLister(MovieLister lister) { this.lister = lister;}`
      - `public List<Movie> findWithDirector(String director) {`
        - `return lister.findMoviesWithDirector(director);`
      - `}`
      - `public static void main(String[] args) {`
        - `new Main().findWithDirector(args[0]); // add code to print list of movies`
      - `}`
    - `}`
- We want this to work even with no explicit call to `setMovieLister();`**

# Two types of dependency injection

---

- In the previous two slides, we've seen (implied) examples of two types of dependency injection
  - **Constructor Injection**
    - `public MovieLister(MoveFinder finder) {`
      - `this.finder = finder;`
      - `}`
  - **Setter Injection**
    - `public void setMovieLister(MovieLister lister) { this.lister = lister;}`
- In the former, the MovieLister class indicates its dependency via its constructor (“I need a MovieFinder”); In the second, the Main class indicated its dependency via a setter (“I need a MovieLister”)

# So, what is dependency injection?

---

- The idea here is that classes in an application indicate their dependencies in very abstract ways
  - MovieLister NEEDS-A MovieFinder
  - Main NEEDS-A MovieLister
- and then a **third party injects (or inserts) a class that will meet that dependency at run-time**
- The “third party” is known as a “Inversion of Control container” or a “dependency injection framework”
  - There are many such frameworks; one example is Spring which has been around in some form since October 2002

# The basic idea

---

- Take
  - a set of components (concrete classes + interfaces)
- Add
  - a set of configuration metadata
- Provide that to
  - a dependency injection framework
- And finish with
  - a small set of bootstrap code that gets access to an IoC container, retrieves the first object from that container by supplying the name of a generic interface, and invokes a method to kick things off

# Example

---

- For instance, Fowler’s example uses the following Spring-specific code to create an instance of MovieLister
- ```
public void testWithSpring() throws Exception {
  - ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");
  - MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
  - Movie[] movies = lister.moviesDirectedBy("Terry Gilliam");}
```
- “spring.xml” is a standard-to-Spring XML file containing metadata about our application; it contains information that specifies that MoveLister needs a TabDelimitedMovieFinder and that the database is in a file called “movies.txt”
 - Spring then ensures that TabDelimitedMovieFinder is created using “movies.txt” and inserted into MovieLister when `ctx.getBean()` is invoked

getBean()?

- In Spring, POJOs (plain old java objects) are referred to as “beans”
 - This is a reference to J2EE’s notion of a JavaBean
 - which is a Java class that follows certain conventions
 - a property “foo” of type String is accessible via
 - `public String getFoo();`
 - and
 - `public void setFoo(String foo);`
- Once you have specified what objects your application has in a Spring configuration file, you pull instances of those objects out of the Spring container via the `getBean` method

Spring's Hello World example

- I shall now possibly horrify you with a “Hello World” example written using Spring
 - I say “horrify” because it will seem horribly complex for a Hello World program
 - The complexity is reduced however when you realize that Spring is architected for really large systems
 - and the “complexity tax” imposed by the framework pays off when you are dealing with large numbers of objects that need to be composed together
 - the “complexity tax” pays dividends when you are able to add a new type of object to a Spring system by adding a new .class file to your classpath and updating one configuration file

Spring's Hello World (I)

- Note: example taken from the Apress book “Pro Spring 2.5”
- First, define a MessageSource class

```
1 public class MessageSource {
2
3     private String message;
4
5     public MessageSource(String message) {
6         this.message = message;
7     }
8
9     public String getMessage() {
10        return message;
11    }
12
13 }
14
```

Spring's Hello World (II)

- Second, define a MessageDestination interface and a concrete implementation

```
1 public interface MessageDestination {  
2  
3     public void write(String message);  
4  
5 }  
6
```

```
1 public class StdoutMessageDestination implements MessageDestination {  
2  
3     public void write(String message) {  
4         System.out.println(message);  
5     }  
6  
7 }  
8
```

Spring's Hello World (III)

- Third, define a MessageService interface

```
1 public interface MessageService {  
2  
3     public void execute();  
4  
5 }  
6
```

Spring's Hello World (IV)

- Fourth, define a concrete implementation of MessageService

```
1 public class DefaultMessageService implements MessageService {
2
3     private MessageSource source;
4     private MessageDestination destination;
5
6     public void setSource(MessageSource source) {
7         this.source = source;
8     }
9
10    public void setDestination(MessageDestination destination) {
11        this.destination = destination;
12    }
13
14    public void execute() {
15        destination.write(source.getMessage());
16    }
17
18 }
19
```

Spring's Hello World (IV)

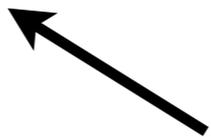
- Fifth, create a main program that gets a Spring container, retrieves a `MessageService` bean, and invokes the service

```
1 import org.springframework.beans.factory.support.BeanDefinitionReader;
2 import org.springframework.beans.factory.support.DefaultListableBeanFactory;
3 import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
4 import org.springframework.core.io.FileSystemResource;
5
6 import java.io.File;
7
8 public class DISpringHelloWorld {
9
10     public static void main(String[] args) {
11         DefaultListableBeanFactory bf = new DefaultListableBeanFactory();
12         BeanDefinitionReader reader = new PropertiesBeanDefinitionReader(bf);
13         reader.loadBeanDefinitions(
14             new FileSystemResource(
15                 new File("hello.properties")));
16
17         MessageService service = (MessageService) bf.getBean("service");
18         service.execute();
19     }
20 }
21 }
22 }
```

Spring Init Code



Where the magic happens



Spring's Hello World (V)

- I say “magic” on the previous slide, because with that call to `getBean()`, the following things happen automatically
 - an instance of `MessageSource` is created and configured with a message
 - an instance of `StdoutMessageDestination` is created
 - an instance of `MessageService` is created
 - the previous two instances (message source, message destination) are plugged into `MessageService`
- In short, you got back an instance of `MessageService` without having to create any objects; and, the object you got back was ready for use
 - you just had to invoke “`execute()`” on it

Spring's Hello World (VI)

- How does the magic happen?

- With the hello.properties file

```
1 source.(class)=MessageSource
2 source.$0=Hello Spring
3 destination.(class)=StdoutMessageDestination
4 service.(class)=DefaultMessageService
5 service.source(ref)=source
6 service.destination(ref)=destination
```

- It defines three beans: source, destination, and service
 - \$0 refers to a constructor argument; (class) sets the concrete class of the bean; (ref) references a bean defined elsewhere
- With this information, the “service” bean can be created and configured

XML Configuration Files

- The use of property files are now deprecated; instead, configuration metadata is stored in XML files; Here's an XML file equivalent to hello.properties:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:lang="http://www.springframework.org/schema/lang"
5       xsi:schemaLocation="
6 http://www.springframework.org/schema/beans http://www.springfr
7 http://www.springframework.org/schema/lang http://www.springfr
8
9   <bean id="source" class="MessageSource">
10     <constructor-arg index="0" value="Hello XML Spring" />
11   </bean>
12
13   <bean id="destination" class="StdoutMessageDestination" />
14
15   <bean id="service" class="DefaultMessageService">
16     <property name="source" ref="source" />
17     <property name="destination" ref="destination" />
18   </bean>
19
20 </beans>
```

Spring's Hello World (VII)

- To use hello.xml, the main program is simplified to:

```
1 import org.springframework.beans.factory.xml.XmlBeanFactory;
2 import org.springframework.core.io.FileSystemResource;
3
4 import java.io.File;
5
6 public class DIXMLSpringHelloWorld {
7
8     public static void main(String[] args) {
9         XmlBeanFactory bf =
10             new XmlBeanFactory(
11                 new FileSystemResource(
12                     new File("hello.xml")));
13
14         MessageService service = (MessageService) bf.getBean("service");
15         service.execute();
16     }
17
18 }
19
```

The Infamous Zoo Example

- Way back in Lecture 4, I mentioned that it was possible to create a version of the Zoo program that would only reference “Animal” and not any of its subclasses (Dog, Cat, Hippo, etc.)
- To do this in Spring, we make use of its ability to specify collection classes in its configuration XML files (see next slide)
 - The main routine is simply a variant on what we’ve seen before
 - we will load a “zoo.xml” configuration file
 - retrieve the “zoo” bean
 - and invoke its “exerciseAnimals()” method

The zoo.xml file

```
8
9 <bean id="zoo" class="Zoo">
10   <constructor-arg index="0">
11     <list>
12       <ref local="bat" />
13       <ref local="cat" />
14       <ref local="dog" />
15       <ref local="elephant" />
16       <ref local="hippo" />
17       <ref local="lion" />
18       <ref local="rhino" />
19       <ref local="tiger" />
20       <ref local="wolf" />
21     </list>
22   </constructor-arg>
23 </bean>
24
25 <bean id="bat" class="Bat" />
26 <bean id="cat" class="Cat" />
27 <bean id="dog" class="Dog" />
28 <bean id="elephant" class="Elephant" />
29 <bean id="hippo" class="Hippo" />
30 <bean id="lion" class="Lion" />
31 <bean id="rhino" class="Rhino" />
32 <bean id="tiger" class="Tiger" />
33 <bean id="wolf" class="Wolf" />
```

Here, we define that there is a bean called “zoo” and it takes a parameter to its constructor that is a list of beans, in this case beans that reference the Animal subclasses below

Here we define instances of animal subclasses; this is where the subclass names are referenced (nowhere else)

Wrap Up

- This represents a barebones introduction to dependency injection frameworks
 - You've seen only a smidgen of Spring's functionality
- But, you've seen the core feature of dependency injection frameworks
 - The ability to remove the names of concrete classes out of your source code while having those classes automatically instantiated and injected into your system based on configuration metadata

Semester Wrap-Up

- Reviewed core OO A&D concepts and techniques
- Covered design patterns and saw examples of how they can be integrated into OO A&D life cycles
 - Saw a wide range of patterns (there are many more out there)
- Covered features of the Android and iOS mobile application frameworks
 - Saw the use of design patterns in these frameworks
- Covered refactoring, test driven design, object-relational mapping and dependency injection
- Provided you an opportunity to build a mobile app and/or web service and make use of design patterns in your prototypes

Coming Up Next Time

- All done!
 - (Well almost, I'll see most of you on Monday for project demos!)
- Have a great winter break!