Presentation by Ed Lance from nTier Training

Materials from Learning Patterns

Overview of the Spring Framework

www.nTierTraining.com

info@nTierTraining.com

Toll Free: 866-526-3921

The Challenge of Enterprise Applications

- **Enterprise applications** have many complex requirements which makes them difficult to write
  - Many object types, with complex dependencies between them
  - Persistent data to be retrieved from and stored to a data store
  - Transactional requirements
  - Remote (distributed) access requirements
  - Web access requirements
- Java/Java EE (Enterprise Edition) has many capabilities to build these enterprise applications
  - Java SE defines basic building blocks like Java primitives and core libraries like Collections
  - Java EE defines a large suite of technologies for building enterprise applications
  - However, there are shortcomings to the Java solution

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Shortcomings of Java/Java EE

- Though rich in functionality, Java/Java EE has significant shortcomings when writing applications, some of which are:
- Basic Java has little support for **managing the lifecycle of your objects and the dependencies** between them
  - Since Java programs are composed of many collaborating objects, managing them requires significant effort
- The Java EE platform is quite large and requires a fairly heavyweight application server
  - Even if you are not using most of the capabilities of Java EE, the application server generally supports all of them
- Java EE has been difficult to use, and intrusive in its design
  - For example, using Enterprise JavaBeans (EJB) required you to write business classes that implemented a specific API
  - This is changing, and newer releases, such as EJB 3, are less intrusive

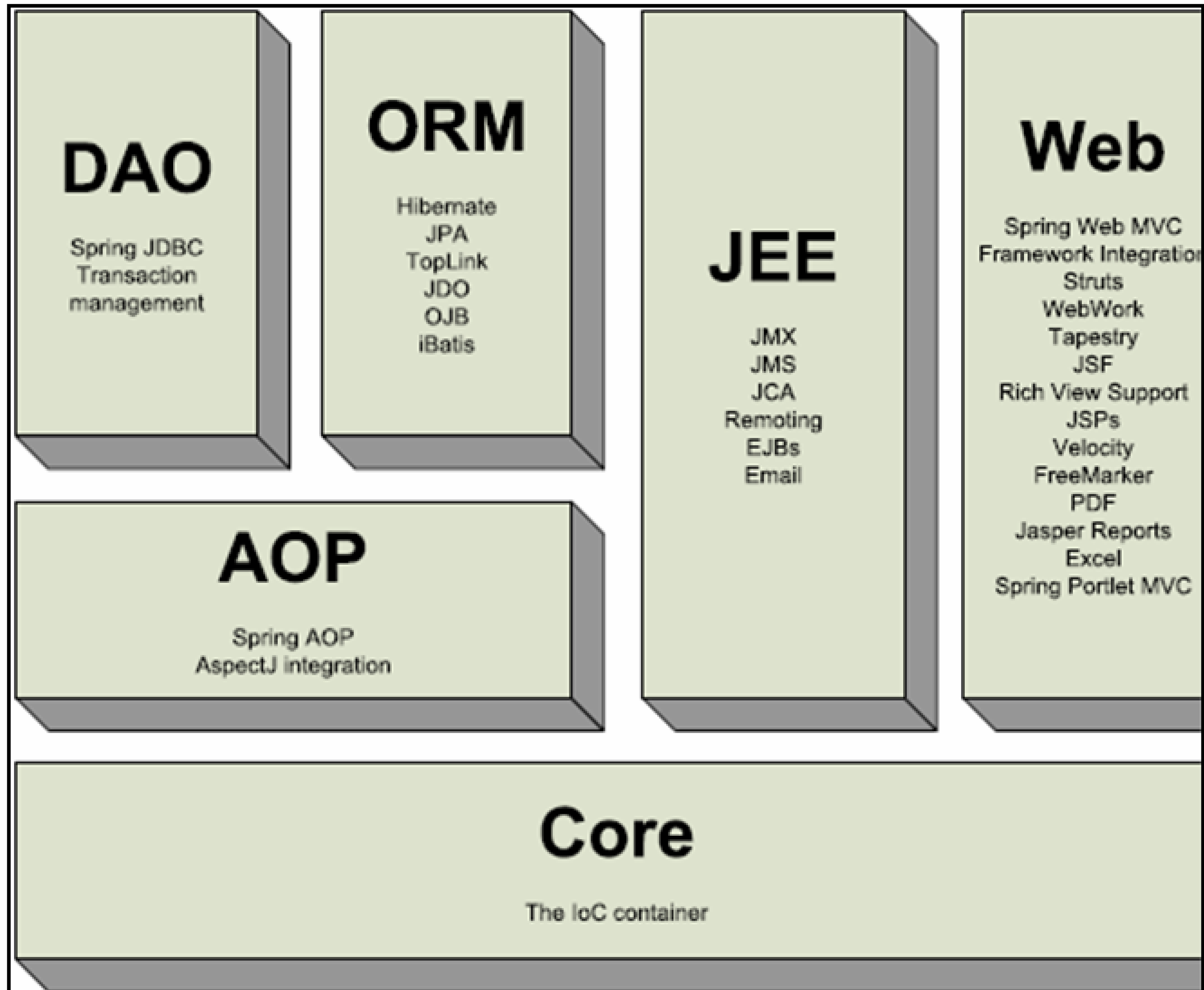
Copyright © 2007 LearningPatterns Inc. All rights reserved.

What is Spring?

- Spring is a **lightweight framework for building enterprise applications**
  - It is "lightweight" in the sense that it is non-intrusive in your programming, and allows you to use only those parts you need
- It provides the following capabilities
  - A **Dependency Injection** (**Inversion of Control**) / container to manage objects and their dependencies
  - A **DAO** package that abstracts JDCB and simplifies its use
  - An **ORM** package providing integration with technologies such as Hibernate and JPA (Java Persistence API)
  - An **AOP** package for doing aspect-oriented programming
  - A **Web** package to integrate with Web technologies
  - An **MVC** package that provides a full Model-View-Controller based Web framework for Web applications

Copyright © 2007 LearningPatterns Inc. All rights reserved.

The Spring Components



Copyright © 2007 LearningPatterns Inc. All rights reserved.

The Spring Distribution

- The Spring home page is **http://www.springframework.org**
  - Spring can be downloaded from this site
- The framework is distributed as a set of zip files
  - Generally speaking, each module is packaged in its own zip
- Spring also has a large number of dependencies on other technologies
  - For example, commons logging, log4j, and many others
- There is a **basic download** which has just the spring jars for all the modules, and a reference manual
  - There is a **download with dependencies** which has all the jars spring depends on, and additional documentation such as the javadocs for the API – it is much larger than the basic download

Copyright © 2007 LearningPatterns Inc. All rights reserved.

LEARNINGPATTERNS

Spring Introduction

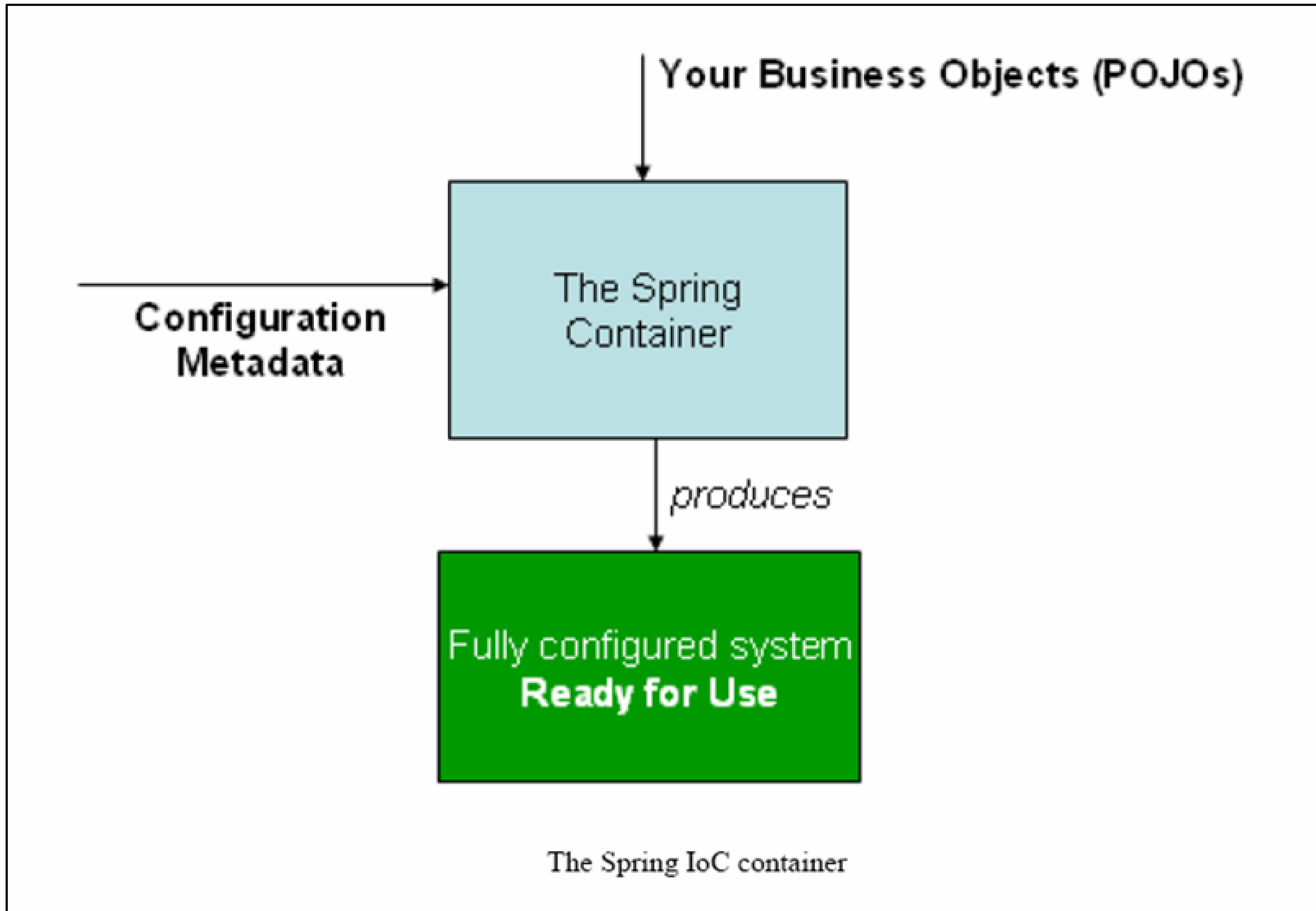
Overview
Spring Introduction
Dependency Injection

Managing Beans

- One of the core Spring capabilities is managing the objects in your application
- These managed objects are called **beans**
  - Spring borrowed this terminology from JavaBeans and EJB
  - A bean is just a **POJO** (Plain Old Java Object)
- Spring uses a **container**, the **Dependency Injection** (**DI**) or **Inversion of Control** (**IoC**) container, to manage objects
  - Configuration metadata is supplied to the container so it knows how to **instantiate, configure, and assemble** your beans
  - Generally, this configuration data is in a simple XML format
  - We will only cover the XML format in this course
- You use the configuration metadata to provide bean definitions and dependencies to the container
  - The container then does all the work of managing the beans

Copyright © 2007 LearningPatterns Inc. All rights reserved.

A Basic Spring Application



The Spring IoC container

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Some Bean Classes

- Let's say we had the following types defined
  - An interface, *Teacher*, defining the functionality we want to use
  - A concrete class, *JavaInstructor*, which implemented this interface, and provided the functionality
- Let's look at how to manage them using Spring

```
package com.javatunes.teach;

public interface Teacher {
	public void teach();
}
```

```
package com.javatunes.teach;

public class JavaInstructor implements Teacher {

	public void teach() {
		System.out.println("BeanFactories are way cool");
	}
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Configuration Metadata

- A **Spring configuration** contains bean definitions
  - They give Spring the information it needs about your beans
  - The bean definitions correspond to actual objects in your application that the Spring container will manage
  - Typically, there are many bean definitions, with dependencies between them
- Beans are configured using ***<bean>*** elements inside a top level ***<beans>*** element, as shown in the example file, *beans.xml*, below

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

        <bean id="springGuru" class="com.javatunes.teach.JavaInstructor"/>

</beans>
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Declaring Beans

- Each bean definition must specify a **package-qualified class name** for the bean
  - This is usually the actual implementation class of the bean (i.e. the class that the container will instantiate)
  - In our example, the ***`class`*** attribute, specifies the implementation as ***`com.javatunes.teach.JavaInstructor`***
  - This tells the container how to instantiate the bean
- In general, a **bean identifier** (or name) is also specified
  - In our example, we used the ***`id`*** attribute, to specify the identifier as ***`springGuru`***
  - This serves as a label for the bean – both in our code and in the configuration files
  - A bean can have multiple names (or aliases)
  - Generally, bean names use camelCase as the convention for bean names (e.g. springGuru)

Copyright © 2007 LearningPatterns Inc. All rights reserved.

The Spring Container

- The Spring container is represented by the interface ***org.springframework.beans.factory.BeanFactory***
  - *BeanFactory* provides a configuration mechanism for managing objects of any nature
  - There are a number of implementations of *BeanFactory* that come with Spring
  - ***XMLBeanFactory***, the most common implementation, uses the XML configuration metadata we just saw for configuration
- The interface ***org.springframework.core.io.Resource*** is used to abstract access to resources (e.g. a config file)
  - Often used to access the XML data to configure a *BeanFactory*
  - ***FileSystemResource*** is a concrete implementation that allows you to easily access resources on the file system
- We'll show a simple example of using these, and cover them in more depth later

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Working With Spring

- At the very highest level, a typical scenario for using Spring includes the following main steps:
- **Create** (XML) **configuration data** for your beans
  - e.g. A file "*beans.xml*" containing the Spring configuration
  - This is the cookbook that tells Spring how to create objects
- **Access the configuration information** in your code using the **Spring resource classes**
  - e.g. Use ***`FileSystemResource`*** to access *beans.xml*
- **Create a bean factory** which gets its configuration data from the resource you created
  - This creates a bean factory tailored to your beans
- **Create beans** using the bean factory
  - e.g. using the ***`BeanFactory.getbean(String)`*** method

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Demo

- Demo Lab 1.2

A Simple Spring Example

- The following code, along with the *beans.xml* configuration file shown earlier, gives a simple example of using Spring
  - We first instantiate a ***FileSystemResource*** to read *beans.xml*
  - We next instantiate an ***XMLBeanFactory*** based on *beans.xml*
  - Lastly, we ask the factory to instantiate the ***springGuru*** bean

```
package com.javatunes.teach;

import org.springframework.core.io.Resource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class TeachMeSpring {
        public static void main(String[] args) {
                Resource res = new FileSystemResource("beans.xml");
                BeanFactory factory = new XmlBeanFactory(res);
                Teacher teacher =
(Teacher)factory.getBean("springGuru");
                teacher.teach();  // Use our bean
        }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Why Bother?

- In this simple example, it looks like we've done quite a bit of work to instantiate a single instance of a single class
  - What is the benefit?
- The main benefit, in this simple example, is that we've **decoupled** our program code **from a dependency on the implementation** class `JavaInstructor`
  - Our code does not need to know the implementation class
  - All we know is that it provides the functionality we need (i.e. implements the `Teacher` interface)
  - We are free to configure the program with any implementation we want, and our code will not change
  - This seems like a simple thing, but it has a lot of benefits, especially when maintaining large systems
- We'll soon see additional capabilities that make Spring useful

Copyright © 2007 LearningPatterns Inc. All rights reserved.

LEARNINGPATTERNS

Dependencies and Dependency Injection

Overview
Spring Introduction
Dependency Injection

Dependencies Between Objects

- Typically, different objects work together in an OO application
  - For example, Object A will use Object B to accomplish a goal
  - In straightforward applications, Object A will often just create an instance of Object B and use it
  - In that case, Object A **depends on** Object B
- Direct dependencies such as this can lead to several undesirable characteristics in large or complex applications
  - **Rigidity**: Hard to make changes because they affect too many other parts of the system
  - **Fragility**: Changes cause unexpected parts of the system to break
  - **Immobility**: It is hard to reuse functionality in another system because modules can't be disentangled from the application
- We'll show an example of a direct dependency, then show an alternative design approach that uses **Dependency Inversion**

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Example of a Direct Dependency

- In the example below*, *JavaInstructor* creates an instance of *SpringCourseBook* directly
  - *JavaInstructor* **depends** on the details in lower level modules
  - If we decide that Java instructors will get the latest information from the Spring Wiki, instead of from a *SpringCourseBook*, *JavaInstructor* code has to be changed

```
public class SpringCourseBook {
        public String getData() {
                return "Dependencies are not so cool";
        }
}
```

```
public class JavaInstructor implements Teacher {
        SpringCourseBook springBook = new SpringCourseBook();
        public void teach() {
                System.out.println(springBook.getData());
        }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Dependency Inversion Principal

- High level or low level modules should not depend upon each other, **instead both should depend upon abstractions**
  - You develop abstractions, for example Java interfaces, that both high level and low level modules depend on
  - High level modules can be written in terms of the abstractions (the interfaces), and not directly on the low level modules
- Using this design strategy has a number of advantages:
  - Allows for the creation of **less coupled components** with a high degree of separation of responsibilities
  - Results in **greater flexibility** since particular implementations can be swapped in and out without affecting other modules
  - Facilitates **reuse** of components since they are less coupled to other parts of an application

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Example of Dependency Inversion

- Both *JavaInstructor* and *SpringCourseBook* depend on ***InfoSource***

```
public interface InfoSource {
        public String getData();
}
```

```
public class SpringCourseBook implements InfoSource {
        public String getData() {
                return "Dependencies are not so cool";
        }
}
```

```
public class JavaInstructor implements Teacher {
        private InfoSource info;
        public void setInfo (InfoSource infoIn) {
      info=infoIn;
  }
        public void teach() {
                System.out.println(info.getData());
        }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Example of Dependency Inversion

- In the example code below, *JavaInstructor* is initialized with a specific implementation of *InfoSource* (an instance of *SpringCourseBook*)
  - But *JavaInstructor* only sees this as the *InfoSource* type
  - It would make no difference to *JavaInstructor* if we initialized it with some other type that implemented *InfoSource* – for example an *EJBCourseBook*
  - We've decoupled the modules, and made them more flexible and easier to use, reuse, and maintain

```
public class TeachMeSpring {

  public static void main(String[] args) {
      SpringCourseBook springBook = new SpringCourseBook();
          JavaInstructor myInstructor = new JavaInstructor();
      myInstructor.setInfo(springBook);
      myInstructor.teach();
  }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Dependency Injection (DI) in Spring

- Using **Dependency Injection**, Spring allows you to abstract dependencies between modules even more
  - The Spring container can **inject** dependencies into a bean and initialize them when it creates the bean
  - It can inject dependencies via constructors, properties in the bean, or arguments to a factory method
- The dependencies are defined in the Spring configuration file
  - The Spring container uses these dependency definitions to initialize the bean
  - You don't need to explicitly initialize dependencies in your code
- Your bean classes are still normal POJO classes
  - The Spring container only requires that you have the appropriate methods needed to initialize the dependencies (e.g. a setter)
  - There is nothing else special about the bean classes

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Dependency Injection Configuration

- The Spring configuration file below uses DI
  - It declares two beans: *springBook* and *springGuru* that use the class definitions we've just seen
  - The **<property name="info" ref="springBook"/>** element tells the container to **initialize** the *info* property of *springGuru* with an instance of *SpringCourseBook*
  - It uses **setter injection** - calling *JavaInstructor.setInfo()*
  - You don't need to write **any** code for the injection to happen

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans … > <!-- Namespace declarations omitted -->

        <bean id="springBook" class="com.javatunes.teach.SpringCourseBook"/>

        <bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
           <property name="info" ref="springBook"/>
        </bean>

</beans>
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Dependency Injection Example

- The example below shows how to use our beans with Spring
  - Notice that all our code is written completely in terms of the **abstract *Teacher* interface**
  - Note also that our code, including the *JavaInstructor* class, **is totally decoupled** from the concrete class *SpringCourseBook*
  - The *JavaInstructor* instance has its *info* property initialized with a *SpringCourseBook* instance by the Spring container
  - The dependency is injected by the container transparently

```
// imports not shown
public class TeachMeSpring {

        public static void main(String[] args) {
                Resource res = new FileSystemResource("beans.xml");
                BeanFactory factory = new XmlBeanFactory(res);
                Teacher teacher =
(Teacher)factory.getBean("springGuru");
                teacher.teach();
        }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

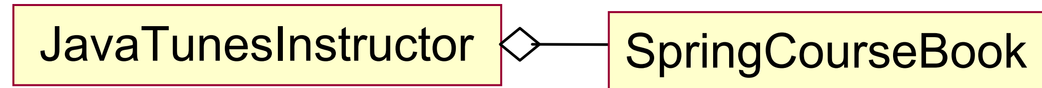
Advantages of Dependency Injection

- DI **reduces the coupling** between modules **in your code**
  - Coupling is basically a measure of the dependencies
- We've seen this reduction in two ways in our example:
  - `JavaInstructor` is not coupled to `SpringCourseBook`
  - `TeachMeSpring` is not coupled to `JavaInstructor` or `SpringCourseBook`
- The dependencies are still there but **not in the code**
  - The dependency is moved to the spring configuration
  - They're injected into beans without your needing to write code
  - This is also commonly referred to as **wiring** beans together
- This leads to **more flexible** code that is **easier to maintain**
  - At a cost – using Spring, and maintaining the spring configuration
  - This is not a trivial cost, but for larger projects the initial cost of adopting spring can be well worth it

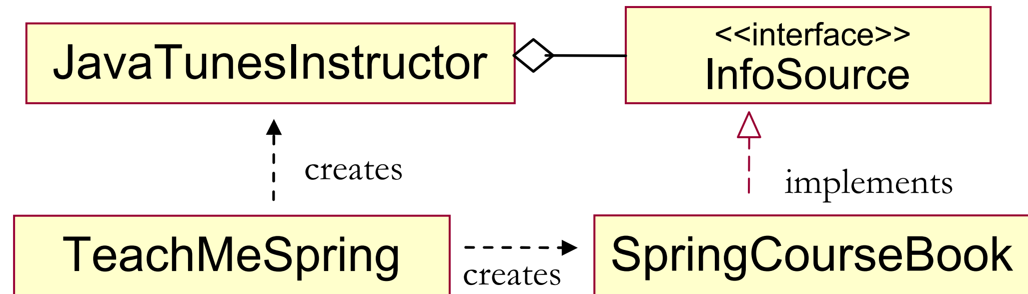
Copyright © 2007 LearningPatterns Inc. All rights reserved.

Dependency Injection Reduces Coupling

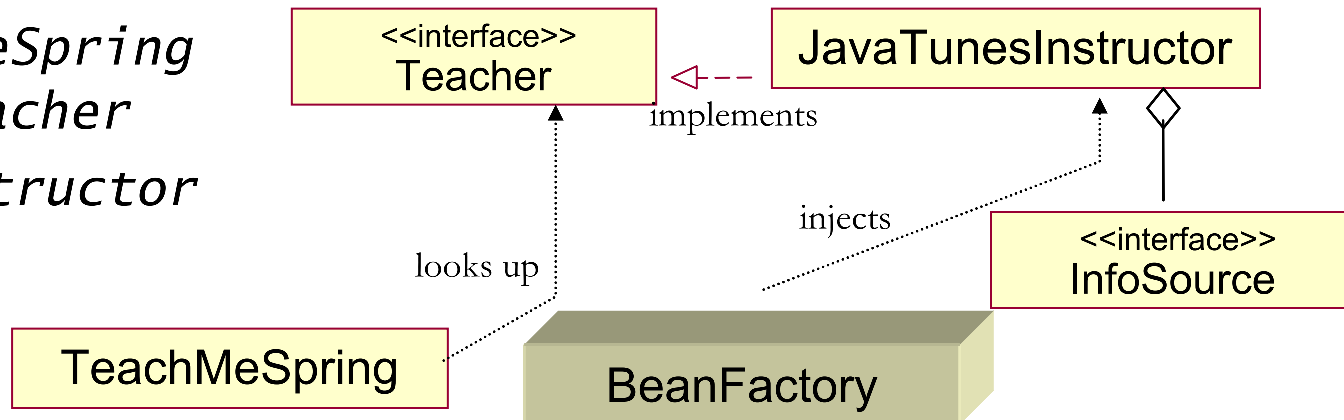
- The simplest case, *JavaTunesInstructor* coupled to *SpringCourseBook*



- *JavaTunesInstructor* coupled to *InfoSource* only
  - *TeachMeSpring* coupled to *JavaTunesInstructor* and *SpringCourseBook*



- Using DI - *TeachMeSpring* only coupled to *Teacher*
  - *JavaTunesInstructor* only coupled to *InfoSource*



Copyright © 2007 LearningPatterns Inc. All rights reserved.

Demo

- Demo Lab 1.2

LEARNINGPATTERNS

Session 7: Spring and the Web

Integration with Java EE

Spring MVC Basics

Forms and View Resolvers

LEARNINGPATTERNS

Integration with Java EE

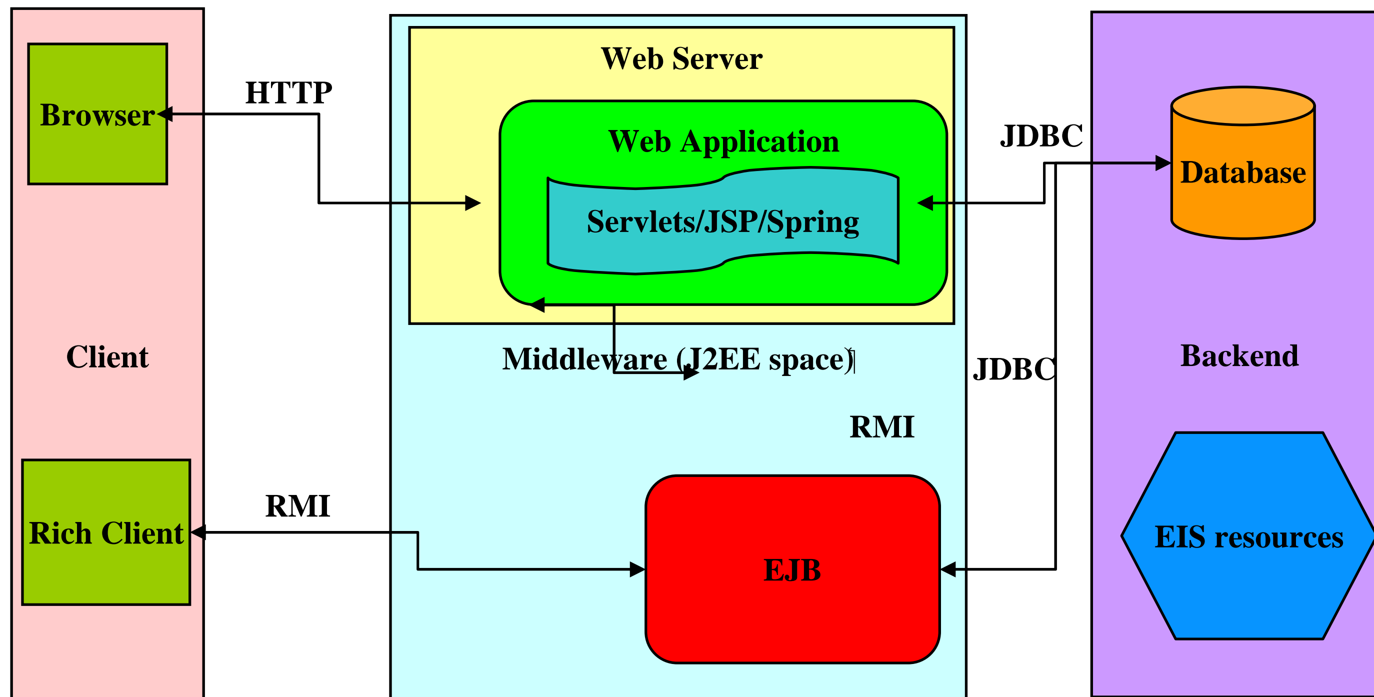
Integration with Java EE

Spring MVC Basics

Forms and View Resolvers

Java EE Web Applications

- Let's start with a brief review of **Web applications** - A standard structure defined in Java EE for organizing applications for the web
  - A collection of **Servlets**, **JavaServer Pages** (JSP), & other files
  - Most frequently collected inside of a Web Archive (WAR) file
    - The standard Java EE way to package up web applications



Copyright © 2007 LearningPatterns Inc. All rights reserved.

Overview

- Page 228 from 117 course

ApplicationContext and Web Apps

- The simplest way to use Spring from within a Web application is to load an application context, and make it available to your Web application
  - Spring has support for doing this very easily
- The ***ContextLoaderListener*** class can be used to load an application context, which can be accessed in your Web apps
  - This listener automatically loads the Spring application context when the web app is deployed
  - It uses a *<context-param>* element in *web.xml* to define the locations of the Spring configuration files
  - To retrieve the application context, use the static method: ***WebApplicationContextUtils.getWebApplicationContext()*** in your web application code
- The following slides shows examples of doing this

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Configuring ContextLoaderListener

- In the *web.xml* file below, we specify two configuration files, *beans.xml* and *javaTunesSearch.xml* with `<context-param>`
  - We then configure the `ContextLoaderListener`

```
<web-app ... >

<context-param>  <!--  config files for ContextLoaderListener -->
  <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/beans.xml
      /WEB-INF/javaTunesSearch.xml
    </param-value>
</context-param>

<listener>  <!-- Load root application context at startup -->
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

</web-app>
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Using the Application Context

- In the servlet definition below, we retrieve the application context using *WebApplicationContextUtils*
  - The context was loaded by the *ContextLoaderListener*
  - We then look up a bean which we can use in our application - just as we did in our earlier non-Web labs
  - The configuration files and *Catalog* beans are from the small non-DB JavaTunes system we used in earlier labs
- You can see - it's very easy to use Spring with Web apps

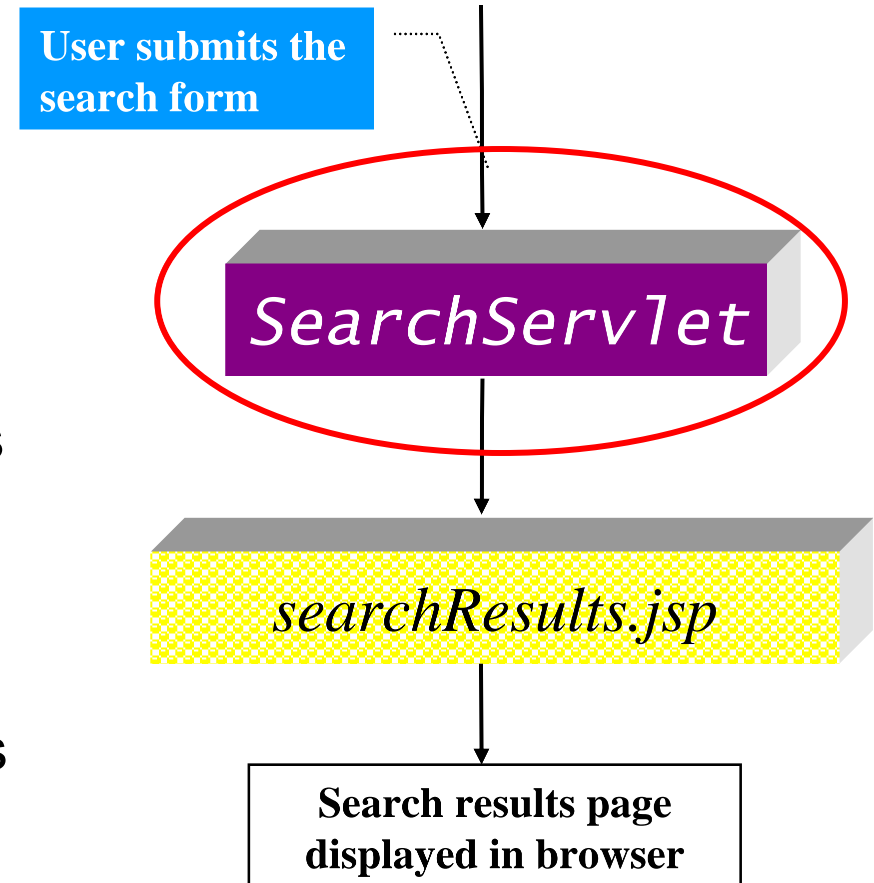
```
import org.springframework.web.context.support.WebApplicationContextUtils;
import org.springframework.web.context.WebApplicationContext;

public class SearchServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {

  WebApplicationContext ctx =
  WebApplicationContextUtils.getRequiredWebApplicationContext(
                                                      getServletContext());
  Catalog cat = (Catalog)ctx.getBean("javaTunesCatalog");
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

The JavaTunes Online Store

Lab

- The Web application we are creating is a small piece of the JavaTunes online music store
  - It displays a search form
  - The search form sends a request to a servlet that does a search and forwards to a results page that displays the results
  - We are working with the servlet processing the search request
  - The flow for JavaTunes appears at right
- The first thing we'll do is set up the environment and the Tomcat Web server



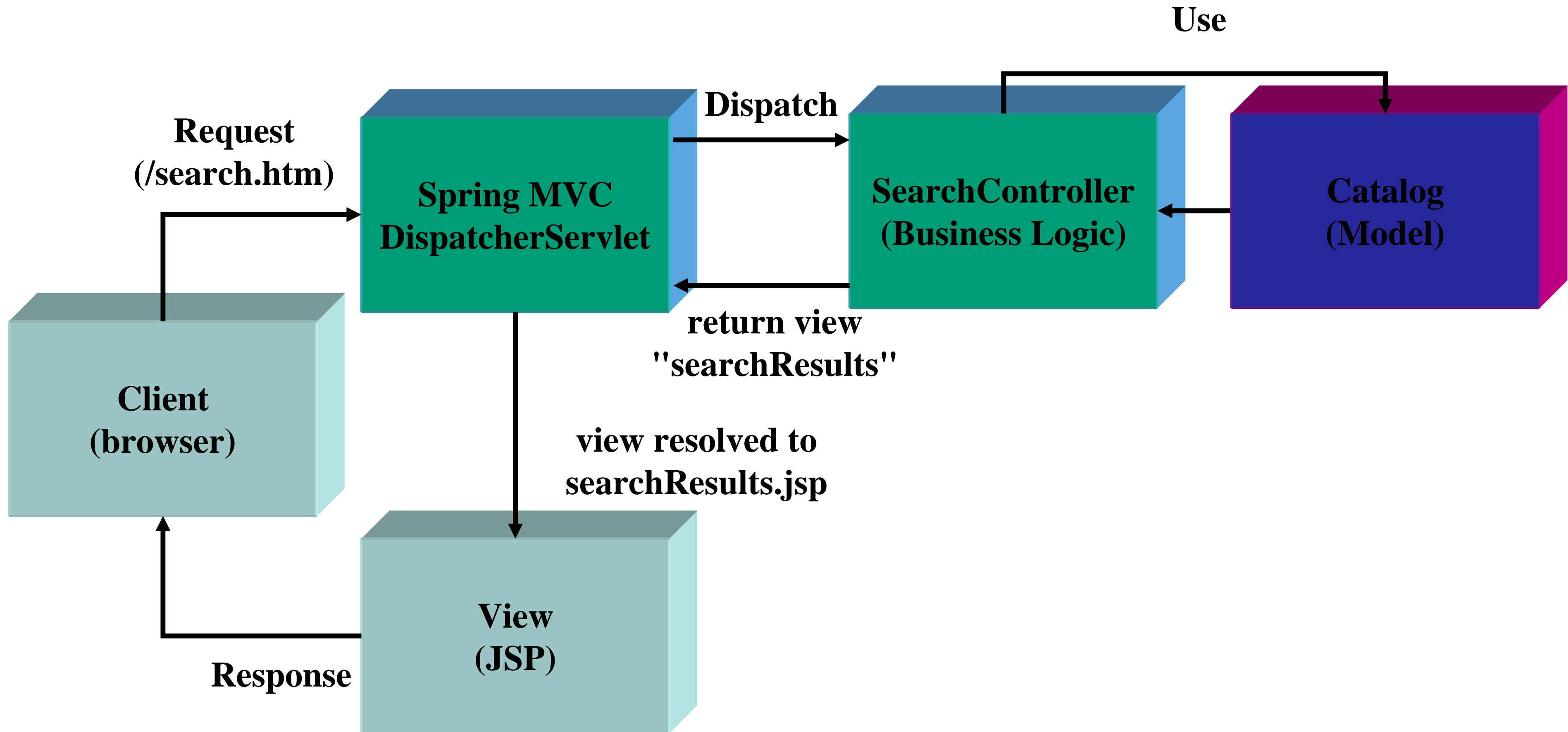
Copyright © 2007 LearningPatterns Inc. All rights reserved.

Spring MVC Architecture

- Spring uses what is called a **front controller** architecture
  - The front controller is a central servlet that receives all requests, and dispatches them to other components (controllers) to handle
  - It is also completely integrated with the Spring container, and allows you to use all its capabilities
- The main components of a Spring MVC application are:
  - ***DispatcherServlet***: The front controller servlet
  - ***Command Controllers***: Contain logic to handle specific user requests, generate the data needed for a response, and return a response to the appropriate view to handle
  - **Handler Mappings**: Map URLs to resources (controllers)
  - **Views**: Generate HTML Output (JSP, other templates)
  - **View Resolvers**: Process response from controllers and direct the application to the appropriate view (e.g. JSP page)
  - **Model**: Spring managed beans

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Simple Search App Model - Spring MVC



Command Controllers

- **Command controllers** are one of the key components of Spring MVC
  - The dispatcher servlet forwards requests to command controllers
  - The command controllers execute business logic (possibly using data from the request) and then return a logical view name to the dispatcher servlet
  - The dispatcher servlet resolves the view to an actual destination (e.g. JSP) which is used to render the view
- We'll look at the simplest command controller first:

```
org.springframework.web.servlet.mvc.AbstractController
```

  - Subclasses use this class by defining the following method to handle the request:

```
ModelAndView handleRequestInternal(
                  HttpServletRequest request,
                  HttpServletResponse response)
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Command Controllers

- The *handleRequestInternal* method is called by Spring to handle the request
  - This method should process the request and return an instance of *org.springframework.web.servlet.**ModelAndView***
  - *ModelAndView* is a simple wrapper class that allows a controller to return a model (holding data for the response) and a view (the resource to use to render the response) in a single value
  - This is made easy to do via the *ModelAndView* constructors, for example, this easy to use one :

```
ModelAndView(String viewName,
             String modelName, Object modelObject)
```

  - The dispatcher servlet looks at the *ModelAndView* instance returned by *handleRequestInternal*
  - It uses the model data to set up the model, and the view name to forward to the resource generating the response

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Very Simple Command Controller

- In the (very simplified) controller below, *SearchController* defines *handleRequestInternal* to:
  - Extract a parameter off the request (the keyword to search for)
  - Get a catalog bean from the spring container
  - Create a *ModelAndView* instance that declares the view as *jsp/searchResults.jsp* *, and includes a model object named *results* that contains the result of the catalog search

```
// imports/package statements not shown
public class SearchController extends AbstractController {

  protected ModelAndView handleRequestInternal(
         HttpServletRequest request, HttpServletResponse response)
         throws Exception {
    String keyword = request.getParameter("keyword");
    WebApplicationContext ctx = getWebApplicationContext();
    Catalog cat = (Catalog)ctx.getBean("javaTunesCatalog");
    return new ModelAndView("jsp/searchResults.jsp",
                              "results", cat.findByKeyword(keyword));
  }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Configuring the Command Controller

- We still need to tell the dispatcher servlet that the previous controller is to be used for handling a request
  - By default, the controller uses a strategy that maps controllers to URLs based on the controllers bean name in the Spring configuration
  - If we want the `SearchController` to handle all requests of the form ***/search.htm***, then we could do so by configuring the bean below in *javatunes-servlet.xml* to have the name `/search.htm`
  - If we had a form like that at bottom, when it was submitted, ***`SearchController.handleRequestInternal()`*** would be called by the dispatcher servlet to handle the request
  - We've now taken the first step in learning Spring MVC

```
<bean name="/search.htm"
           class="com.javatunes.web.SearchController"/>
```

```
<form method='post' action='/javatunes/search.htm'>
  <input size='20' type='text' name='keyword'/><br/>
  <input type='submit' name='Submit' value='Search'/>
</form>
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

A JavaBean Command Class

- Spring MVC command classes are used to hold form data and possibly model data *
  - Below we define a command class to hold our search keyword, and the results collection
  - As you can see, there is nothing special about it - it just has two properties keyword (a *String*) and results (a *Collection*)

```
package com.javatunes.web;
import java.util.Collection;

public class SearchCommand {
  private String keyword;
  public String getKeyword() {return keyword;}
  public void setKeyword(String keyword) {this.keyword = keyword;}

  private Collection results;
  public Collection getResults() {return results;}
  public void setResults(Collection results) {this.results=results; }
}
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

HandlerMappings

- The handler mapping strategy defines how the *DispatcherServlet* will map a URL to a controller (handler)
  - Until now, we've been using the default handler mapping strategy of *BeanNameUrlHandlerMapping*
  - We can configure other strategies, but configuring the associated bean in *javatunes-servlet.xml*
  - For instance, we can use a *SimpleUrlHandlerMapping* to do the same thing as shown below (e.g. if our *SearchController* bean was named *searchContoller*, not *search.htm*)

```
<bean class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <value>
      /**/search.htm=searchController
      /**/checkout.htm=checkoutController
    </value>
  </property>
</bean>
```


Copyright © 2007 LearningPatterns Inc. All rights reserved.

Lesson Summary

- The Spring container can be integrated with normal Web applications using Springs ***`ContextLoaderListener`***
  – This listener automatically loads Spring's root application context and makes it available to Web applications
- **MVC** is a standard pattern for creating applications with user interface requirements
  – It organizes functionality into three areas
  – The **model** represents the business domain; the **view** presents the data to users; the **controller** is an intermediary between the model and the view
- Spring uses Spring beans as its model, command controllers as its controllers and normal views (such as JSP) to generate output
  – It also uses the *`DispatcherServlet`* as a front controller to dispatch incoming requests to Spring MVC

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Lesson Summary

- Spring's ***DispatcherServlet*** is configured in *web.xml* like any other servlet
  - The only difference is that its url-mapping uses patterns, such as *.htm, and is applicable to a whole category or requests
- **Command controllers** handle incoming requests
  - They use data from the request, execute business logic, set up the model with data needed by the view, and indicate what resource should be used to render the view
- **Form controllers** are sophisticated command controllers that process form requests
  - They can extract method parameters into JavaBeans (Spring command classes), and have convenient methods to do normal form processing

Copyright © 2007 LearningPatterns Inc. All rights reserved.

Resources

- The **Spring documentation**
  - The reference manual, API docs, and samples
  - They're all available in the full Spring download
- **http://www.springframework.org**
  - The home of the Spring Framework
- **Spring in Action** - Second Edition by Craig Walls
  - An excellent book, full of useful examples and explanations
  - Its big - 700 pages +

- There are many more - these are a great place to start

Copyright © 2007 LearningPatterns Inc. All rights reserved.