Spring 3.x and Spring MVC/REST

Workshop Overview

- This is an in-depth course covering the use of the Spring 3 framework to build enterprise applications using Java

- The course, at a high level, covers the following areas of Spring technology
  - **Core features of Spring**
  - **Data Access Features**
  - **Transaction Support**
  - **Web Application Support including Spring MVC**
  - **Spring Security Overview**
  - **Building RESTful Services with Spring**
  - **Aspect Oriented Programming (AOP)**

Workshop Objectives

- To understand the Spring framework and use its capabilities, including:
- **Spring Core**: Dependency Injection (DI) and lifecycle management of application objects
  - Using the Spring API and Spring configuration to write Spring programs
  - Configuration covers both XML and annotations
- **Data Access**: Data access via Spring's data support
  - DataSource support
  - Hibernate and JPA-based DAOs
- **Transactions**: Control transactions declaratively with Spring
  - Use the Spring annotations and XML configuration elements
- **Web**: Integrate Spring with Web applications and understand the basic structure and use of Spring MVC
- **Security**: Understand the architecture and basic use of Spring Security
- **REST**: Understand REST principals, and build REST services with Spring
- **AOP**: Use Spring's AOP (Aspect Oriented Programming) capabilities to inject crosscutting concerns

Workshop Agenda

Module 1: **Introduction to Spring**

Module 2: **More About Bean Properties**

Module 3: **The Spring Container and API**

Module 4: **Database Access With Spring**

Module 5: **Transactions**

Module 6: **Web Applications with Spring MVC**

Module 7: **Spring Security Overview**

Module 8: **RESTful Services with Spring**

Module 9: **Aspect Oriented Programming (AOP)**

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

  JavaTeacher teacher = new JavaTeacher()

  – Code fragments use the same font, e.g. *teacher.teach()*

  – We **bold/color** text for emphasis

  – Filenames are in italics, e.g. *JavaInstructor.java*

  – We sometimes denote more info in the notes with a **star ***

  – Longer code examples appear in a separate code box - e.g.

```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
   public void teach() {
      System.out.println("BeanFactories are way cool");
   }
}
```

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - Many follow a common fictional case study called **JavaTunes**
    - An online music store
  - The labs are in the course book, and include detailed instructions
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

Release Level

Lab

- This manual has been tested with, and contains instructions for, running the labs using the following platforms:
  - **Spring 3.2** (tested with 3.2.2)
    - The material is compatible with Spring 3.1 (see notes)
  - **Spring Security 3.1** (tested with 3.1.3)
  - **Java 6 or 7**
  - **Eclipse Java EE Edition** (any fairly recent edition - tested with 4.2/Juno)
  - **Tomcat 7.0.x** for the Spring/Web material
- Recent similar versions of the software will likely work except for potentially small configuration changes

Module 1: Introduction to Spring

Overview

Spring Introduction

Dependency Injection

Lesson Objectives

- Understand why we need the Spring Framework
- Understand what Spring is, and how it simplifies enterprise application development with Java
- Learn how Spring uses configuration information and Dependency Injection (DI)
  - To manage the beans (objects) in an application
  - To manage bean dependencies

Overview

Overview

Spring Introduction

Dependency Injection

The Challenge of Enterprise Applications

- **Enterprise applications** have many complex requirements
  - Many types, with complex dependencies
  - Persistent data retrieved from / stored to a data store
  - Transactional requirements
  - Remote (distributed) access requirements
  - Web access requirements
- Java / Java EE (Enterprise Edition) provides many capabilities to build enterprise applications
  - Java SE: Basic building blocks like Java primitives and core libraries like Collections
  - Java EE: Enterprise technologies like EJB, Servlets/JSP, JDBC
  - However, there are shortcomings to the standard Java solution

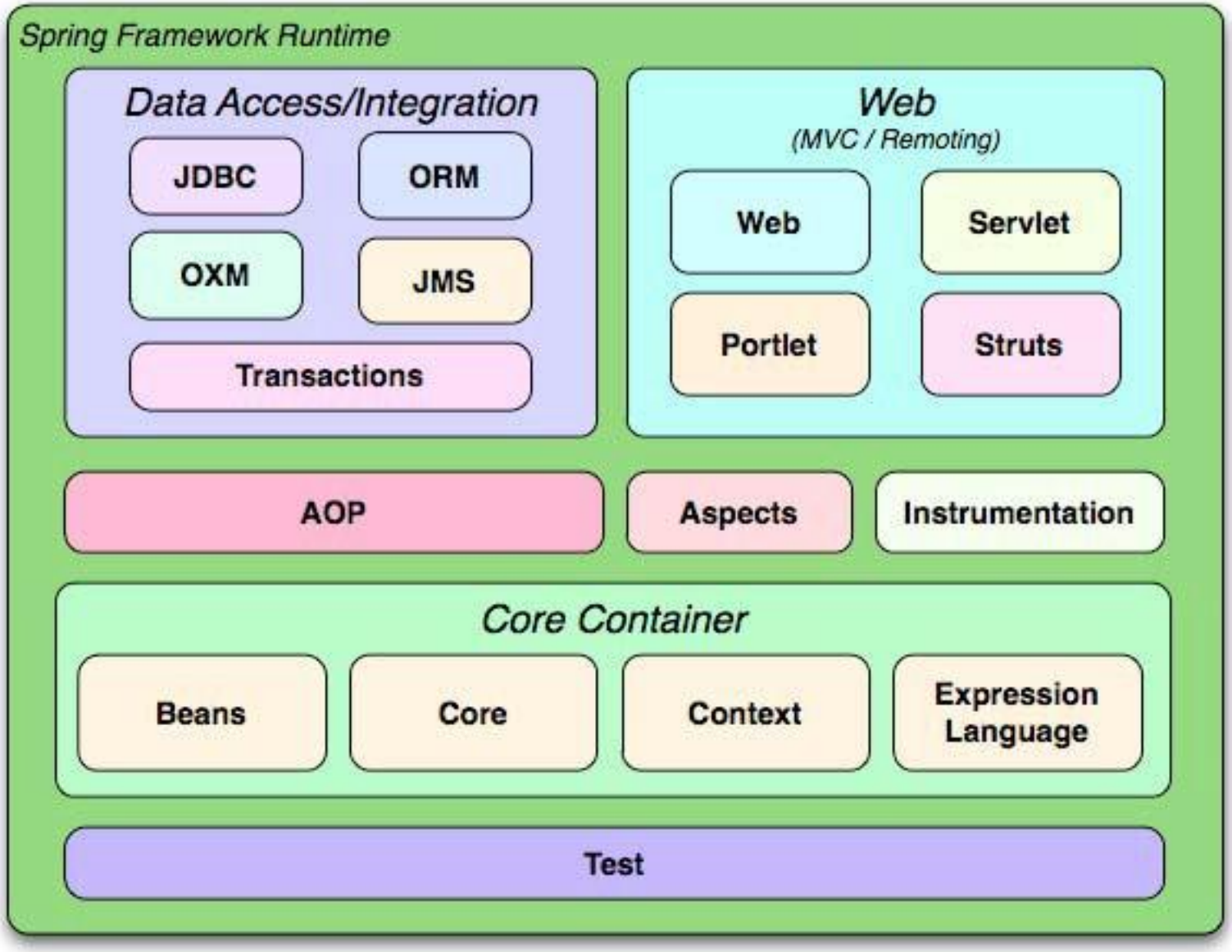
Shortcomings of Java/Java EE

- Java/Java EE can have significant shortcomings, including:
- Little support for **managing the lifecycle of your objects and the dependencies** between them
  - Managing large numbers of objects requires significant effort
- Java EE is large and requires a fairly heavyweight app server
  - Even if using only a subset of capabilities (e.g. Web apps and transactions), the application server generally supports all of them
- Java EE has been difficult to use, and intrusive in its design
  - Not as big an issue now
  - Newer releases, such as EJB 3, are less intrusive

What is Spring?

- **Lightweight framework to build enterprise applications**
  - "Lightweight" meaning non-intrusive in your programming, and allowing you to use only what you need
- It includes the following capabilities
  - A **Dependency Injection** (**Inversion of Control**) container to manage objects and their dependencies
  - A **DAO** package that abstracts JDBC 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
  - A **security** framework for authentication and authorization

The Spring Modules

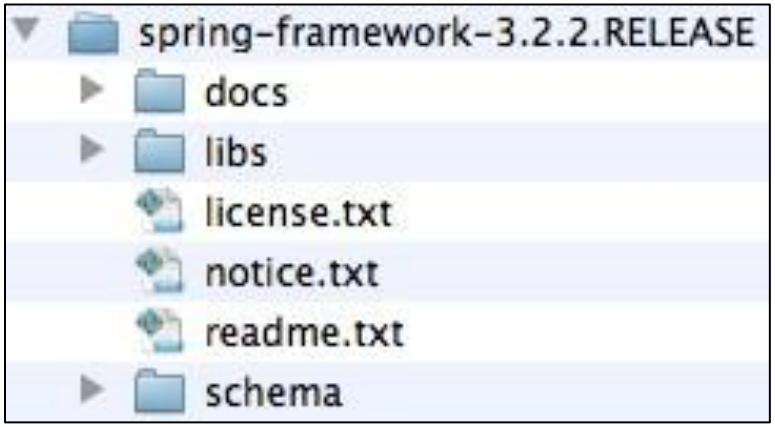


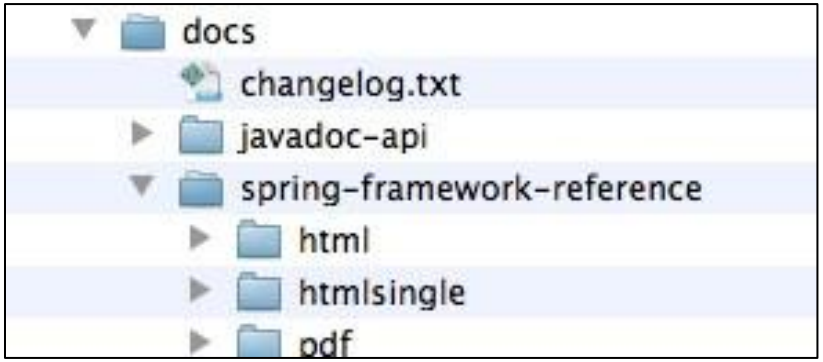
The Spring Distribution

- Spring home page: **http://www.springsource.org/**
  - Can download Spring from this site
- Framework distributed as a set of jar files
  - Each module packaged in its own archive
    - e.g. *spring-beans-3.2.2.RELEASE.jar*
  - The core Spring download has the all the spring module jars, a reference manual, and API docs
- Spring has many dependencies on external technologies
  - e.g. - Apache commons logging, jars for AOP, and more
  - **Not included** in the core Spring download
- **Maven** is commonly used to manage Spring's dependencies
  - Project management tool - can automatically download needed dependencies
  - Maven covered more later

The Spring Distribution

- Below left, is the folder structure of the core distribution
  - Below right, are some of the jars in the libs folder
    - Note that there are source and javadoc jars also
  - At bottom is the docs folder





Lab 1.1 – Setting Up the Environment

Lab 1.1 – Set up the Environment

Lab

- **Overview**: In this lab, we will become familiar with the lab structure, set up our working environment, and test it to make sure it is running well
  - The end goal is to compile and run an application that requires the Spring libraries
- **Objectives**:
  - Become familiar with the lab structure
  - Set up our environment
  - Write and run a simple application that requires Spring
- **Builds on previous labs**: None
- **Approximate Time**: 45-60 minutes

Information Content and Task Content

Lab

- Within a lab, information only content is presented in the normal way – the same as in the student manual pages
  - Like these bullets at the top of the page
- Tasks that the student needs to perform are in a box with a slightly different look – to help you identify them
- An example appears below

Tasks to Perform

- Look at these instructions, and notice the different look of the box as compared to that above
  - Make a note of how it looks, as future labs will use this format
- OK – Now **get out your setup files**; we're ready to start working

Setup Environment

Lab

Tasks to Perform

- Make sure Java is installed
  - Usually in a folder (for Java 7) like *C:\Program Files\Java\jdk1.7.0_nn*
  - If not, then install it - it can be downloaded from:

  http://www.oracle.com/technetwork/java/javase/downloads/index.html

- Check that Spring 3.2 is installed – likely in a folder like:

  C:\spring-framework-3.2.2.RELEASE
  - It is not required - you can use it for reviewing the documentation
  - We supply the needed jars in the setup
  - It can be downloaded from **http://www.springframework.org/download**

Extract the Lab Setup Zip File

Lab

- To set up the labs, you'll need the course setup zip file *
  - e.g.:

  LabSetup_Spring3x_Tomcat-Eclipse_ .zip

- Our base working directory for this course will be ***C:\StudentWork\Spring***
  - This directory will be created when we extract the Setup zip
  - It includes a directory structure and files (e.g., Java files, XML files, other files) that will be needed in the labs
  - All instructions assume that this zip file is extracted to C:\. **If you choose a different directory, please adjust accordingly**

Tasks to Perform

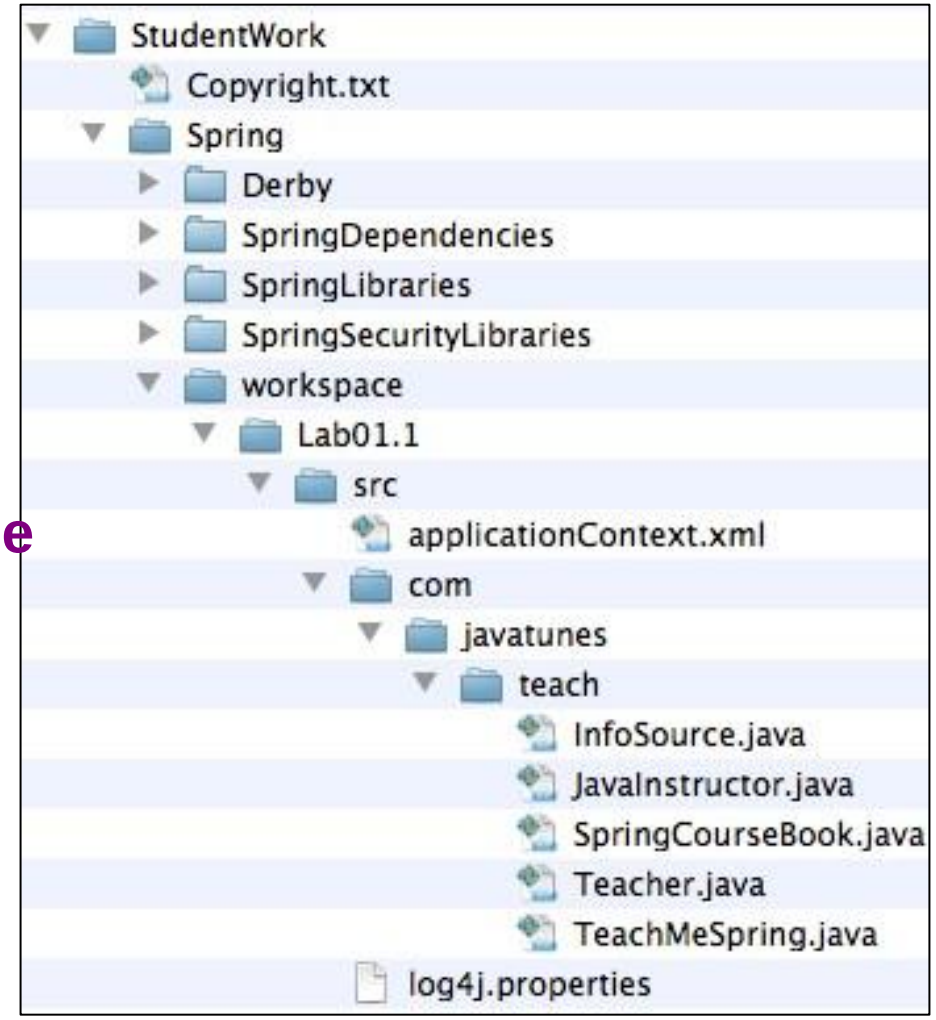
- Unzip the lab setup file to ***C:***
  - This will create the directory structure, described in the next slide, containing files that you will need for doing the labs

Lab Directory Structure

Lab

- **StudentWork\Spring** contains
  - **Derby**: Database files
  - **SpringDependencies:** jars for Spring dependencies
  - **SpringLibraries:** Spring jars
  - **workspace**: Eclipse workspace

- **StudentWork\Spring\workspace** will contain the following folders:
  - **common**: shared files (if any)
  - **LabNN** : Folder for Lab NN
  - **LabNN/src** : Java source
  - **LabNN/bin/** : compiled code (Eclipse's standard location)



General Instructions

Lab

- Root lab folder for this lab is:

  C:\StudentWork\Spring\workspace\Lab01.1

  –Already exists in your workspace

  –You'll create a new Eclipse project in this folder, which contains starter files that we provide

  –In general, all the lab files are under this project/folder and instructions are given relative to this project/folder

- Detailed instructions are included in this lab

  –Including complete **Eclipse** instructions, as well as details about the lab requirements

- Subsequent labs are similar to this lab, so include fewer detailed instructions

The Eclipse Development Environment

Lab

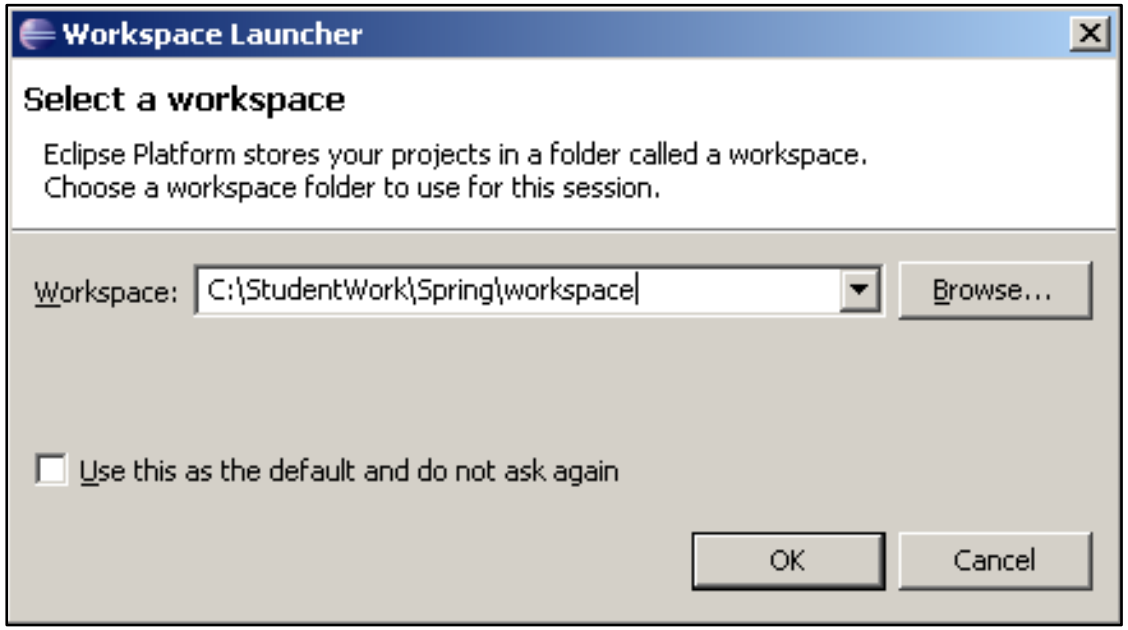
- **Eclipse**: Open source platform for building integrated development environments (IDEs)
  - Used mainly for Java development
  - Can be flexibly extended via plugins to add capabilities
  - **http://www.eclipse.org** is the main website
- This lab includes detailed instructions on using Eclipse
  - Starting it, creating and configuring projects, etc.
- Other labs include fewer Eclipse details - they may just say build/run as previously
  - Just use the same procedures to build/run as in this lab
  - Refer back to these lab instructions as needed

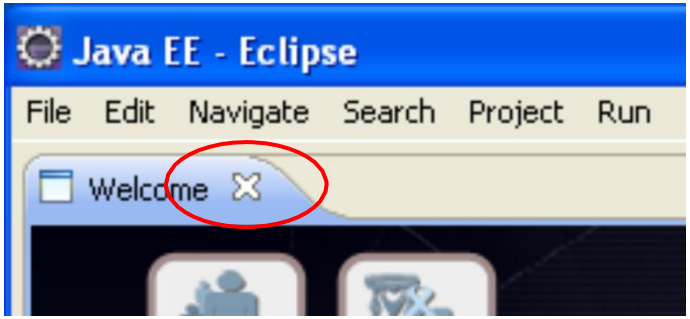
Launch Eclipse

Lab

Tasks to Perform

- Launch eclipse: Go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box appears prompting for workbench location (below left)
  - Set the workbench location to **C:\StudentWork\Spring\workspace**
  - If a different default Workbench location is set, change it
  - Click **OK**
  - **Close** the Welcome screen: Click the X on its tab (below right)



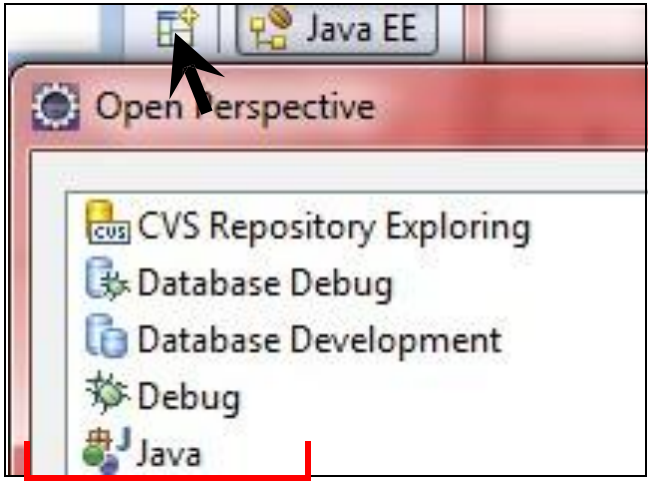


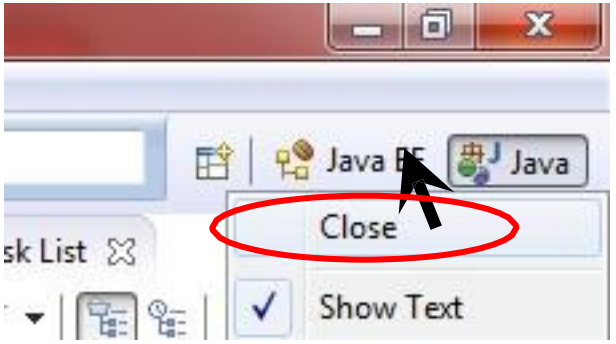
Workbench and Java Perspective

Lab

Tasks to Perform

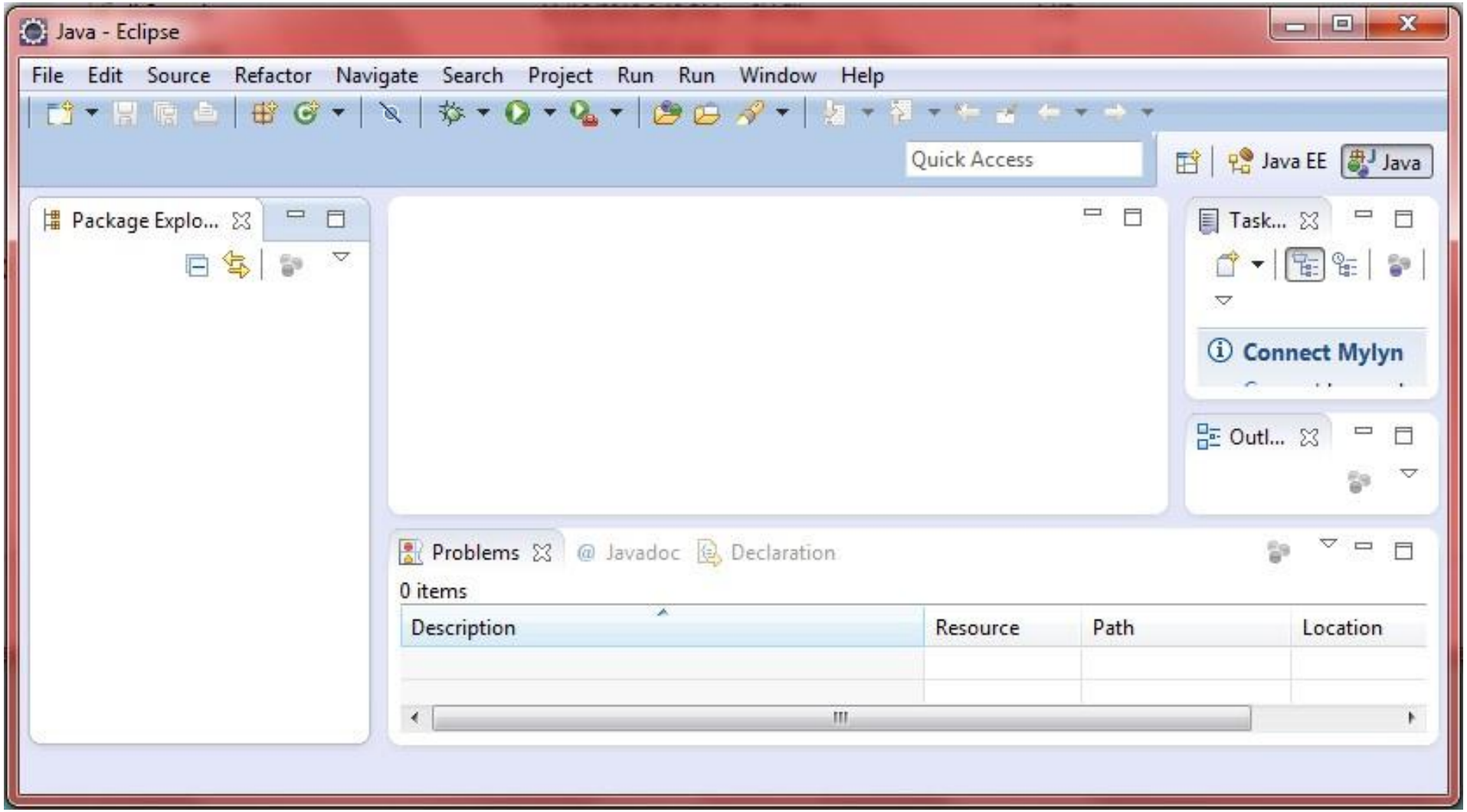
- You should be in a Java EE perspective *
- **Open a Java Perspective: Click the Perspective icon** at the top right of the Workbench, and select Java (see below left)
  - Close the Java EE perspective by right clicking its icon, and selecting close (see below right)
- Unclutter the Java Perspective by closing some views
  - **Close** the Task List, Outline and Hierarchy views (click on the X)
  - Open the Navigator View (**Window | Show View | Navigator**)





The Java Perspective

Lab



Create User Libraries

Lab

- We'll need to include a number of Spring jars in our classpath
  - We include the Spring jars from the Spring distribution in the setup
  - We also supply other external dependencies in a separate folder
  - We'll set these up as a user library for ease of use

Tasks to Perform

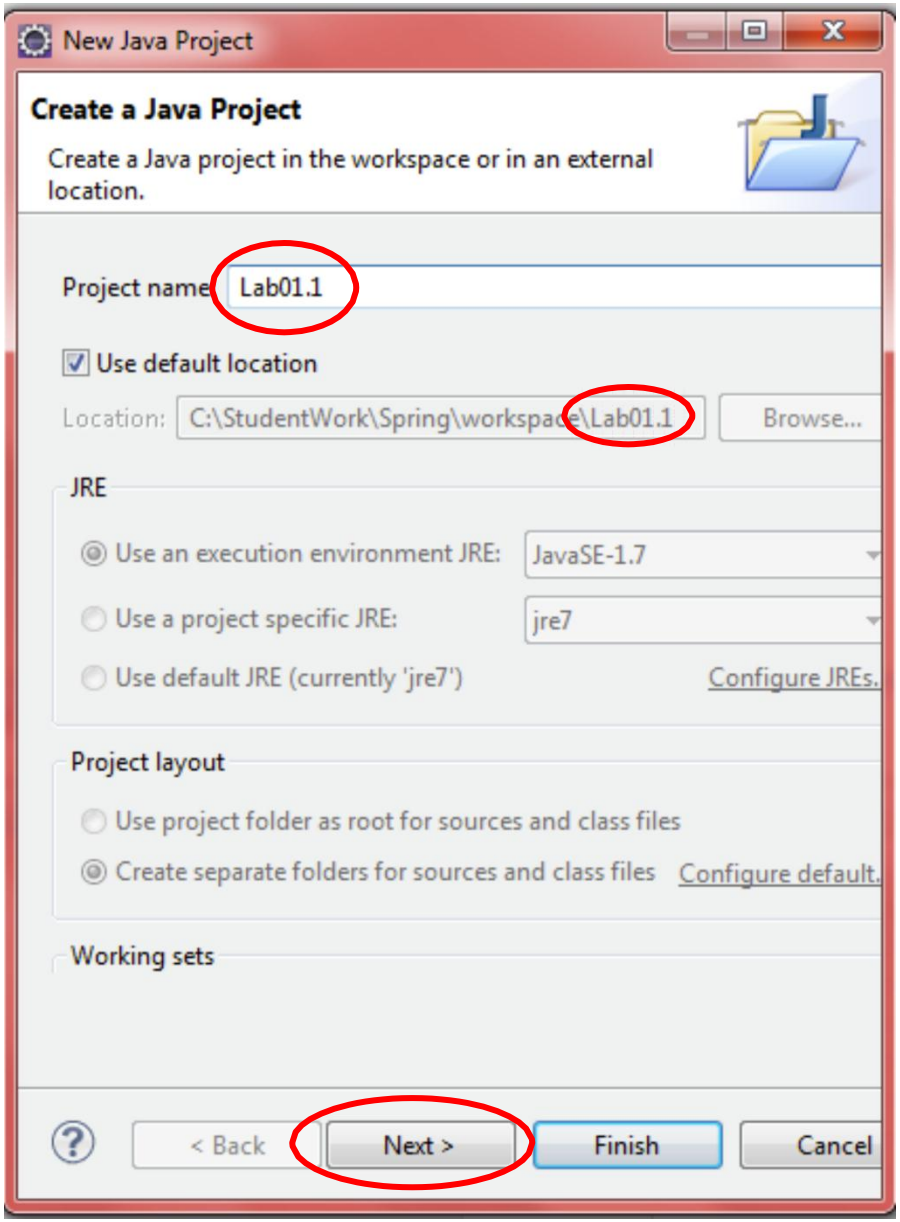
- Go to **Window | Preferences | Java | Build Path | User Libraries**
  - Click **New…** , in the next dialog, call the library **Spring**, and press **OK**
  - Click the **Add External JARs…** button, browse to *StudentWork\Spring\SpringLibraries*, select all the jars in that folder, and click **OK** (see notes)
  - Select the Spring library again, click **Add External JARs…**, browse to *StudentWork\Spring\SpringDependencies,* add in all jars in that folder
- Creates a library with the Spring distribution and dependency jars
  - We'll use it to configure the project build path

Create a Project for our Application

Lab

Tasks to Perform

- Create a **Java Project** *
  - Call the project **Lab01.1**
  - Eclipse will then automatically set the project folder to *Lab01.1*
    - This folder already exists in the workspace
    - It contains a Java project laid out for Eclipse
  - Click **Next**
  - This will bring you to a dialog where you can set the Java Settings for the project

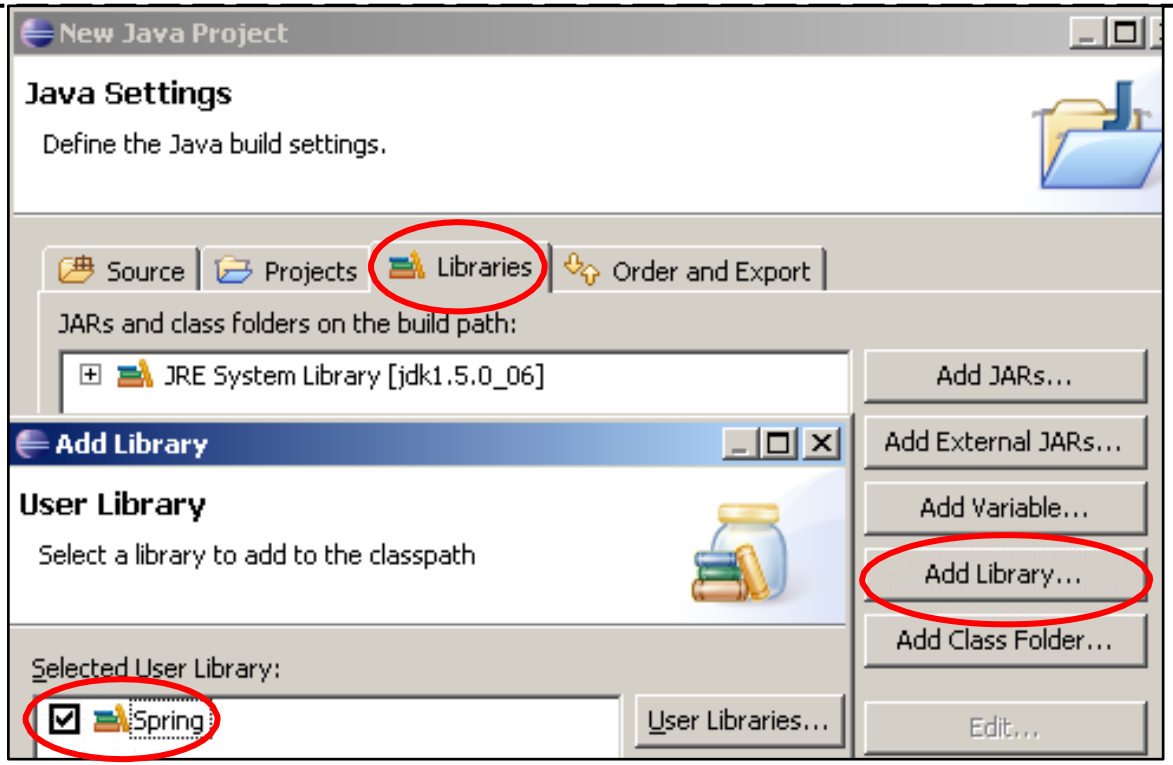


Add the Spring Library to the ClassPath

Lab

Tasks to Perform

- In the **Java Settings** dialog, click the **Libraries** tab *
  - Click the **Add Library** button, and in the dialog that comes up, select **User Library** then click **Next**
  - Check off **Spring**, then click **Finish** in all open dialogs *

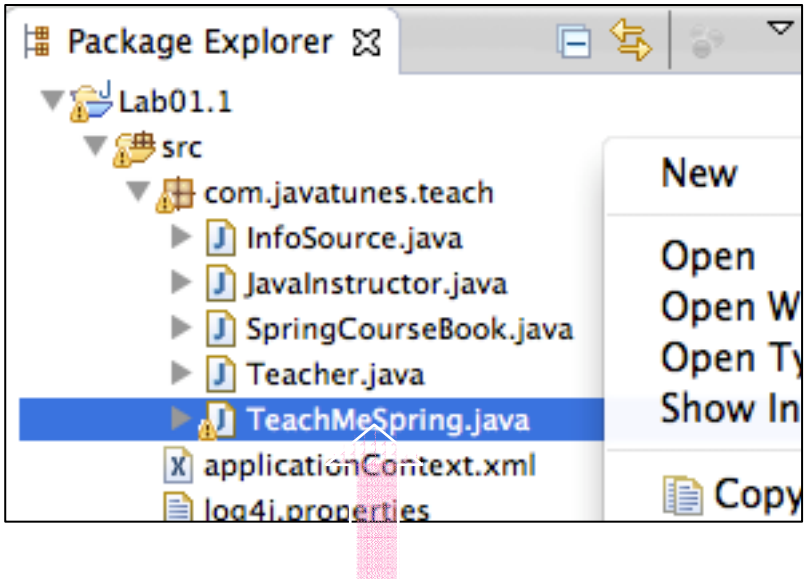


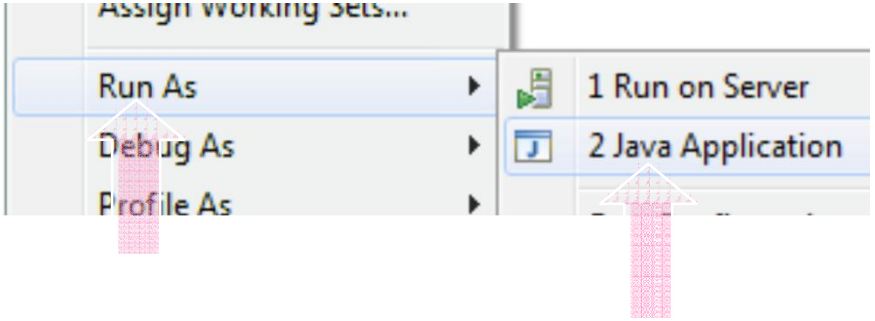
Running the Lab

Lab

Tasks to Perform

- We will use Eclipse to run a skeleton program we supply in *TeachMeSpring.java*
- After a clean build (one that is error-free, but not necessarily warning-free), do the following
  - In the **Package Explorer** pane, right click on *TeachMeSpring.java* (in package com.javatunes.teach under the src folder)
  - Select: **Run As -> Java Application**
  - This will automatically find the main method in *TeachMeSpring* and run it for you

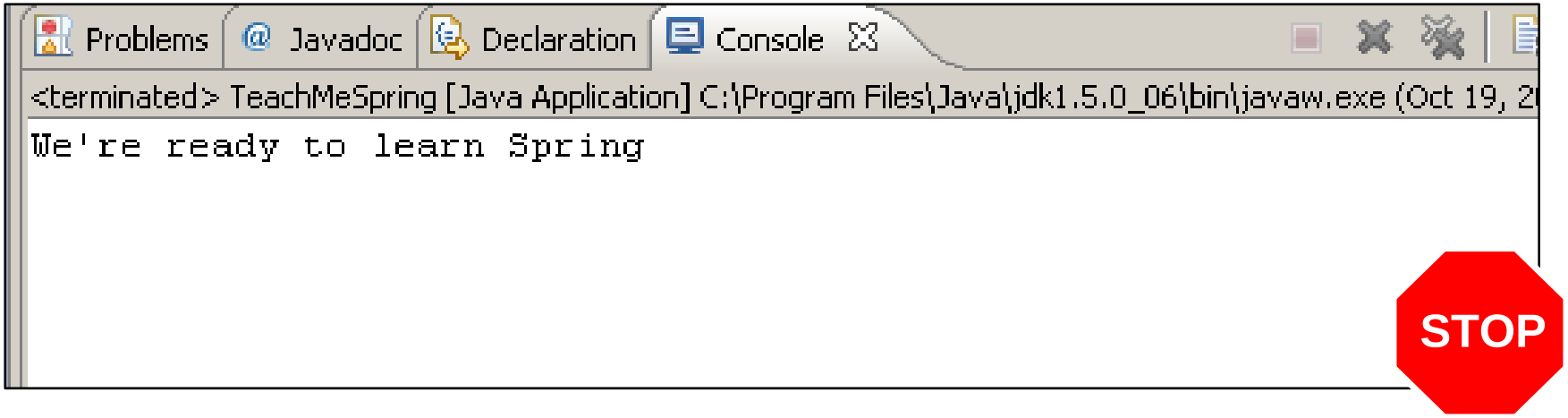




Program Output

Lab

- You should see output in the Eclipse Console view showing the results of running the program
  - Similar to that shown below
  - If you see any exceptions, then you have a problem somewhere
- **Note**: You will follow similar procedures **whenever you have a Java program to run** in the labs
  - The only thing that may change is the program class that you'll need to right click on to run



Spring Introduction

Overview

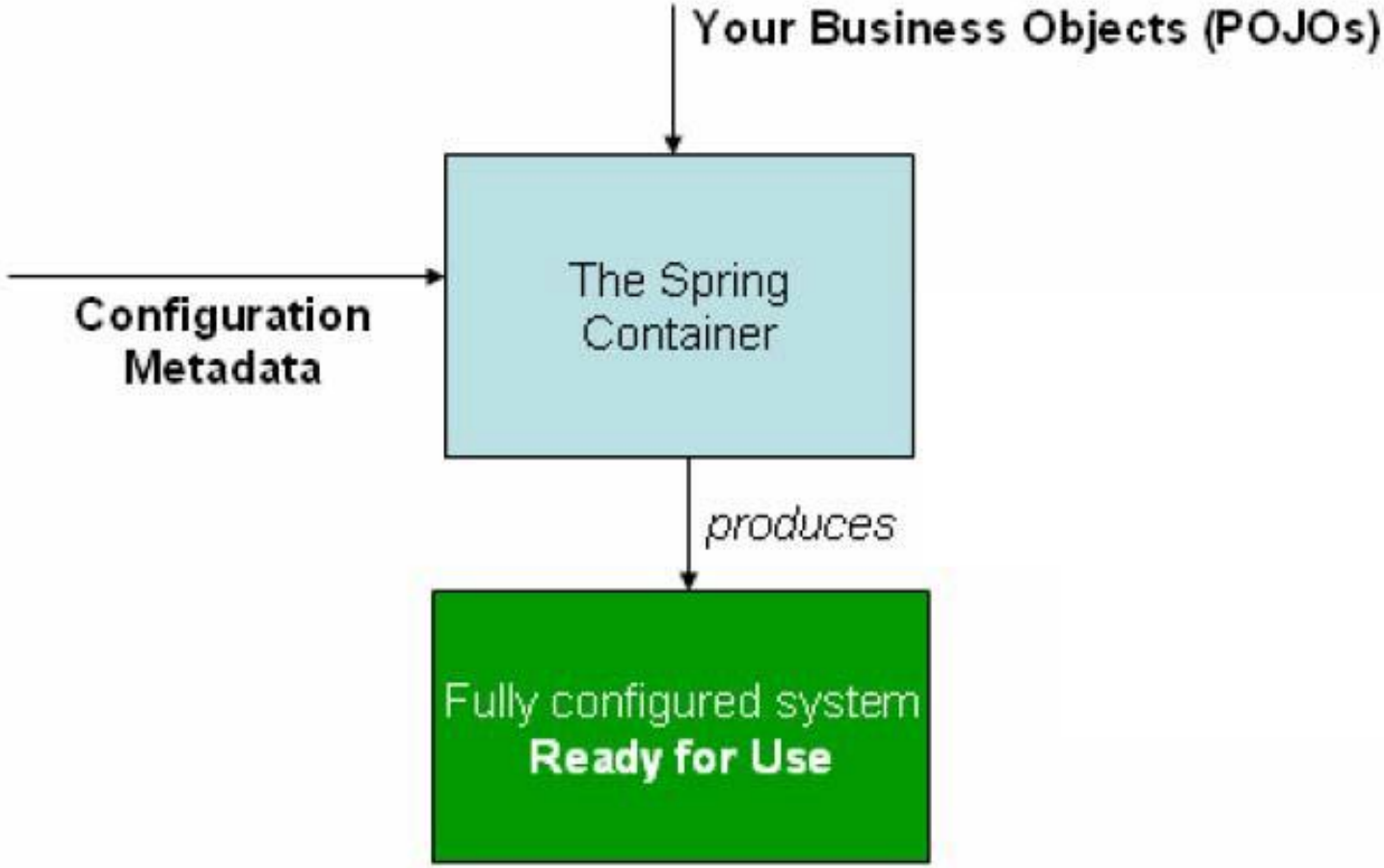
Spring Introduction

Dependency Injection

Managing Beans

- **Managing application objects** is a core Spring capability
- These managed objects are called **beans**
  - Just a **POJO** (Plain Old Java Object)
  - Bean is a common name (e.g. JavaBeans and EJB)
- The Spring **container** manages objects
  - Often called the **Dependency Injection (DI)** or **Inversion of Control** (**IoC)** container
  - **Configuration metadata** is used by the container to **instantiate, configure, and assemble** your beans
  - Generally supplied in simple XML format
- You write a configuration with **bean definitions and dependencies**
  - The container does the work of managing the beans

A Basic Spring Application



The Spring IoC container

Some Bean Classes

- Assume the following types (your business objects)
  - ***Teacher*** interface defining our functionality
  - ***JavaInstructor***: Concrete class, implementing *Teacher*
  - Note: Spring strongly advocates **programming to interfaces** (see notes)
- Let's look at how to manage instances of our types with 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");
   }
}
```

Configuration Metadata

- A **Spring configuration** contains bean **definitions**
  - And information about your beans (the metadata)
  - Bean definitions correspond to actual objects in your app that the Spring container will manage
  - Typically - many bean definitions and dependencies
- Beans are configured with a top level ***<beans>*** element
  - It contains namespace declarations and ***<bean>*** elements as shown in the *applicationContext.xml* file below

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

<!-- The beans namespace is the default one for the document -->
<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-3.1.xsd">

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

</beans>
```

Declaring Beans

- Each bean definition must specify a **package-qualified class name**
  - Usually the bean implementation class (i.e. the class that the container will instantiate)
  - In our example, the ***class*** attribute, specifies the implementation as ***com.javatunes.teach.JavaInstructor***

- In general, a **bean identifier** (or name) is also specified
  - In our example, the ***id*** attribute, specifies the identifier as ***springGuru***
    - It is a label for the bean – both in code and the config file
  - A bean can have multiple names (or aliases)
  - Generally, bean names use camelCase for bean names (e.g. springGuru)

The Spring Container

- Spring container instantiates/configures application objects
  - Provides a configuration mechanism for managing objects
  - And resolves object dependencies based on configuration
- *org.springframework.context.***ApplicationContext**
  - Core API to **access the Spring container** in your code
  - Multiple implementations provided for flexibility, e.g.
  - ***ClassPathXmlApplicationContext***: Commonly used implementation that loads resources from the class path
- *ApplicationContext* extends interface *org.springframework.beans.factory.**BeanFactory***
  - Defines many core methods, but usually not used directly
  - *ApplicationContext* is the recommended API choice

Working With Spring

- High level scenario for using Spring includes the following:

- **Create Spring configuration data** for your beans
  - e.g. In an XML file like *applicationContext.xml*
  - Or via annotations in code (covered later)
  - The "cookbook" telling Spring how to create objects

- **Initialize Spring container, create access point**, e.g.
  - **Create an application context** instance that reads config data
    - It will initialize the beans in the config file
  - Use the context instance to access beans

- **Retrieve beans** via the context and use them
  - e.g. use ***getBean()*** to look up a bean by name
  - *getBean()* is part of *BeanFactory*/*ApplicationContext* API

A Simple Spring Example

- The code below provides a simple Spring example
  - Based on *applicationContext.xml* seen earlier
  - Instantiates a ***ClassPathXmlApplicationContext*** to read *applicationContext.xml* from the classpath to create the spring container
  - Looks up the ***springGuru*** bean from the container and uses it
  - Closes the context (destroying all beans) *

```
package com.javatunes.teach;

import
    org.springframework.context.support.ClassPathXmlApplicationContext;

public class TeachMeSpring {
   public static void main(String[] args) {
      ClassPathXmlApplicationContext ctx = new
           ClassPathXmlApplicationContext("applicationContext.xml");
      // Note that getBean uses Java generics (no casting needed)
      Teacher teacher = ctx.getBean("springGuru", Teacher.class);
      teacher.teach();    // Use our bean
      ctx.close();   // Close the context
   }
}
```

Why Bother?

- The example looks like a lot of work to create a single object !

- Main benefit: We've **decoupled** our code **from a dependency on the implementation** class *JavaInstructor*
  - Our code does not need to know about *JavaInstructor*
  - We just know about the functionality we need
    - i.e. The *Teacher* interface (which *JavaInstructor* implements)
  - We can use any conforming implementation in our configuration
    - Our client code **will not change**
  - This can be very useful for complex systems

- We'll soon see additional capabilities that make Spring useful

Some BeanFactory/ApplicationContext API

- ***boolean containsBean(String)***: returns true if *BeanFactory* contains a bean definition or instance with the given name
- ***<T> T getBean(String, Class<T> requiredType)***: Returns a bean instance, registered under the given name
  - Spring 3 style using Java generics (see notes) and is preferred
- ***Object getBean(String)***: Old style – non generics version
  - Requires cast
- ***Class<?> getType(String name)***: Returns the Class of the bean with the given name
- ***boolean isSingleton(String)***: Determines whether or not the named bean is a singleton
- ***String[] getAliases(String)***: Return the aliases for the given bean name, if any
- Many more methods - view the javadoc for more details

Lab 1.2 – Hello Spring World

Lab 1.2 – Hello Spring World

Lab

- **Overview**: In this lab, we will create and use a Spring context to access a bean instance
  - The end goal is to build a complete (and simple) Spring application and run it
  - This lab uses the *JavaInstructor*, *Teacher*, and *TeachMeSpring* types shown previously in the student manual
- **Objectives**:
  - Become familiar with the different parts of a basic Spring program
  - Write and run a simple Spring application
- **Builds on previous labs**: 1.1
- **Approximate Time**: 20-30 minutes

Writing and Configuring a Bean in Spring

Lab

Tasks to Perform

- Continue to work in the **Lab01.1** project
- Open up the *JavaInstructor.java* file
- The class is in the *com.javatunes.teach* package, under ***src***
- Make sure *JavaInstructor* implements the Teacher interface
  - Remember that we code to interfaces to decouple us from a specific implementation
- Open up the *src/applicationContext.xml* file
  - Click the **Source tab** to edit it
  - Finish up the declaration of the <*bean*> element by declaring:
    - An id of *springGuru*
    - A class of *com.javatunes.teach.JavaInstructor*

Using a Bean in a Program

Lab

Tasks to Perform

- Open up *TeachMeSpring.java*, look for the TODO comments, and add in the following code
  - First, create a *ClassPathXmlApplicationContext* to read *applicationContext.xml*
  - Next, look up the *springGuru* bean from the context
  - Call *teach()* on the *springGuru* bean
  - Finally, call *close()* on the context
- Make sure you have no compilation errors in your code
- **Run** the *TeachMeSpring* program again
  - Run the same way as in previous labs
  - You should see the output from the *teach()* method
- You've successfully configured and used a bean with Spring
  - Congratulations !

Logging and Additional Things (Optional)

- We've provided a *log4j.properties* file in *src*
  - This configures some of the logging that Spring will do
  - You can reduce the logging output by opening this file, and changing the INFO to WARN at the top line of the file:

```
log4j.rootLogger=WARN, stdout
```

  - You can increase the logging by using DEBUG instead

- Other things to try
  - Try looking up the bean by a different name – what happens?
  - Try casting the bean you look up to a *JavaInstructor* instead of a *Teacher*
    - Does this work?
    - Is it a good idea? (see notes)
  - Change your code back to your original solution before going on

Dependencies and Dependency Injection

Overview

Spring Introduction

Dependency Injection

Dependencies Between Objects

- Different objects work together in an OO application
  - e.g., Object A directly uses Object B to accomplish a goal
  - Object A **depends on** Object B

- Direct dependencies can lead to unwanted traits
  - **Rigidity**: Hard to make changes - they affect many other parts of the system
  - **Fragility**: Changes cause unexpected failures in other system areas
  - **Immobility**: Hard to reuse functionality elsewhere - modules can't be disentangled

- We'll show a direct dependency example, then show an alternative approach using **Dependency Inversion**

Example of a Direct Dependency

- Assume *JavaInstructor* uses *SpringCourseBook*
  - Below, *JavaInstructor* creates a *SpringCourseBook* instance directly
  - *JavaInstructor* **depends** on the lower level module details
  - If the data source changes - e.g. to a *SpringWiki*, then *JavaInstructor* code must change (see notes)

```
public class SpringCourseBook { // package statements not shown *
   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());
   }
}
```

Dependency Inversion Principal

- High level or low level modules should **should depend upon abstractions** and not upon each other
  - **Use abstractions**, e.g. **interfaces**, that all modules depend on
  - High level modules are written in terms of abstractions (the interfaces), and not directly in terms of low level modules
  - Dependencies exist, but not exposed in implementation classes

- Using this design strategy has a number of advantages:
  - Facilitates **less coupled components**
    - With a high degree of separation of responsibilities
  - Produces **greater flexibility**
    - Implementations can be swapped without affecting other modules
  - Facilitates **reuse** of components
    - They're less coupled to other parts of an application

Example of Dependency Inversion

- Both *JavaInstructor* and *SpringCourseBook* depend on ***InfoSource*** (an interface) – neither depend on the other

```
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());
   }
}
```

Example of Dependency Inversion

- Below, *JavaInstructor* is initialized with a *SpringCourseBook* instance (that implements *InfoSource*)
  - *JavaInstructor* **only sees this** as the *InfoSource* type
  - We could initialize *JavaInstructor* with any type that implemented *InfoSource* – e.g. a *SpringWiki*
  - 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();
  }
}
```

Dependency Injection (DI) in Spring

- With **Dependency Injection**, Spring lets you abstract dependencies even more
  - The Spring container can initialize and **inject** dependencies into a bean
    - Via constructors, properties in the bean, or arguments to a factory method
- Dependencies **are defined in the Spring configuration**
  - The Spring container uses these dependency definitions to initialize the bean
  - **No need to explicitly initialize dependencies** in your code
- Your bean classes are normal POJO classes
  - You only need appropriate setter methods or constructors needed to initialize the dependencies
  - There is nothing else special about the bean classes

Dependency Injection Configuration

- The Spring configuration below uses DI
  - Declares two beans: *springBook* and *springGuru*
  - *<property name="info" ref="springBook"/>* tells the container to **initialize** the *info* property of *springGuru*
    - Using the instance of *SpringCourseBook* defined above it
  - Uses **setter injection** - calling *setInfo()* on *springGuru*
  - You don't need to write **any** code for the injection to happen

```
<?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>
```

Dependency Injection Example

- The example below shows how to use our beans with Spring
  - Our main code **is decoupled** from *JavaInstructor* - it is written in terms of the ***Teacher* interface**
  - *JavaInstructor*, **is totally decoupled** from the concrete class *SpringCourseBook*, it's written in terms of ***InfoSource***
  - The *JavaInstructor* instance's *info* property is injected with a *SpringCourseBook* instance **by the Spring container**
    - This is transparent to all your code

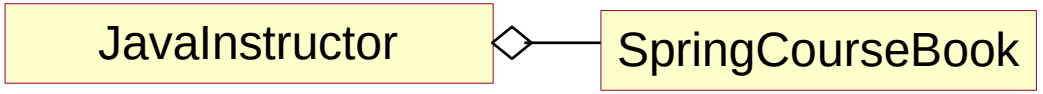
```
// imports not shown
public class TeachMeSpring {

   public static void main(String[] args) {
      ClassPathXmlApplicationContext ctx = new
            ClassPathXmlApplicationContext("applicationContext.xml");
      Teacher teacher = ctx.getBean("springGuru", Teacher.class);
      teacher.teach();
   }
}
```

Advantages of Dependency Injection

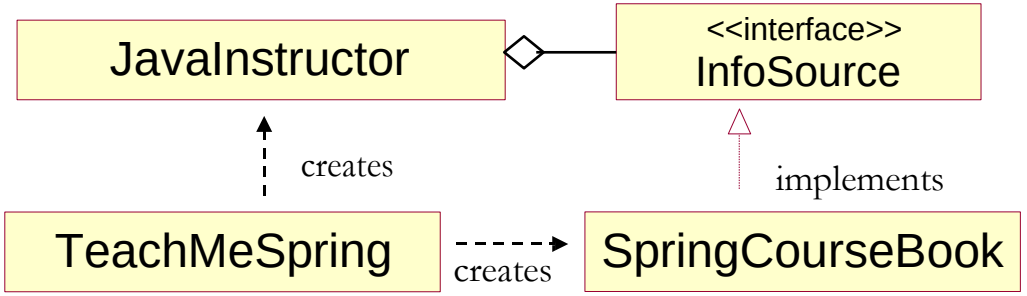
- DI **reduces coupling** between modules **in your code**
  - Coupling is basically a measure of the dependencies
- We've seen this reduction in two ways in our example:
  - *JavaInstructor* is not coupled to *SpringCourseBook*
  - *TeachMeSpring* is not coupled to *JavaInstructor* or *SpringCourseBook*
- The dependencies are still there but **not in the code**
  - Dependencies are **moved to the spring configuration**
  - They're injected into beans without your needing to write code
  - 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

Dependency Injection Reduces Coupling

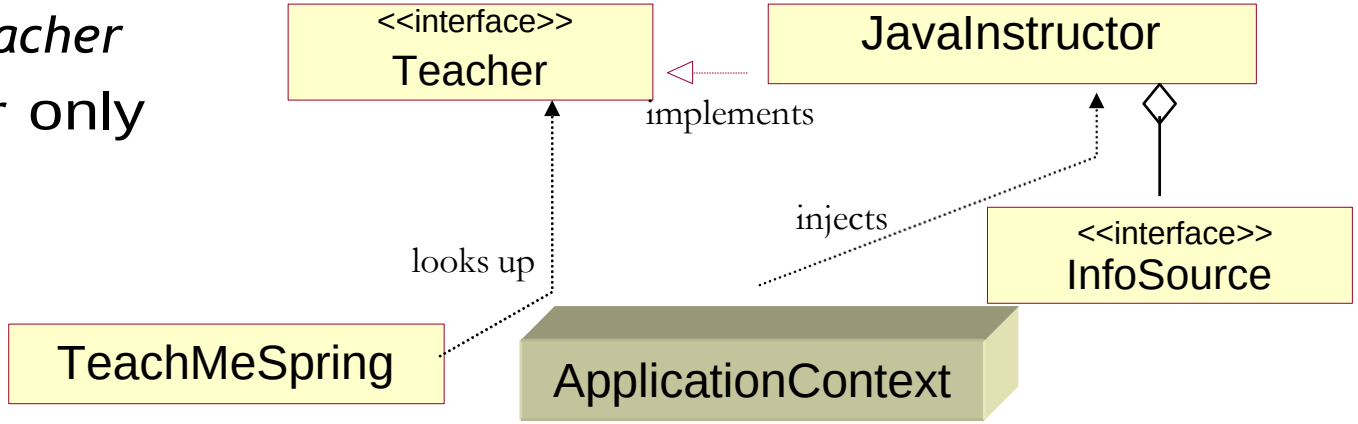
- The simplest case, *JavaInstructor* coupled to *SpringCourseBook*



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



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



Lab 1.3 – Dependency Injection

Lab 1.3 – Dependency Injection

Lab

- **Overview**: In this lab, we'll work with Spring's DI capabilities
  - We will add a property (a dependency on another bean) to the *JavaInstructor* class, and initialize it using Spring DI
  - The main program will not change at all
  - Again, the program is kept very simple so the concepts will be clear
- **Objectives**:
  - Become familiar with Spring DI, and use it in a program
- **Builds on previous labs**: 1.2
- **Approximate Time**: 20-30 minutes

Writing and Configuring a Bean in Spring

Lab

Tasks to Perform

- Continue to work in the **Lab01.1** project
- Open up *SpringCourseBook* for editing
  - Make sure that *SpringCourseBook* implements the *InfoSource* interface
- Open up *JavaInstructor* for editing
  - Add in a property called *info*, of type *InfoSource*
  - You'll need an instance field of type *InfoSource*, as well as a getter/setter method for this field
  - Modify the *teach()* method to print out the data from the *info* properties *getData()* method
- We won't initialize the info property – that will be done with DI

Using a Bean in a Program

Lab

Tasks to Perform

- Open up the *src/applicationContext.xml* file
  - Add in a new bean element for *SpringCourseBook*
    - Give it an id of *springBook*
  - Modify the <*bean*> element for *springGuru* by initializing its *info* property with a <*property*> element
    - Use the *springBook* bean to initialize the *info* property
- **Run** the program as before
  - The output should now be whatever data the *SpringCourseBook* returns
  - The main program doesn't change at all – the *JavaInstructor* dependency on *SpringCourseBook* is completely hidden
  - *JavaInstructor* is also decoupled from *SpringCourseBook*

Additional Things to Try (Optional)

Lab

- In *applicationContext.xml*, try leaving out the initialization of the *springGuru* bean's info property
  - Run your program
  - What happens? Why?

- Change your code back to your original solution before going on

STOP

Review Questions

- What is Spring?
- How does Spring help you to build enterprise applications?
- What is Dependency Injection, and how does it work in Spring

Lesson Summary

- **Spring** is a lightweight framework for building enterprise applications whose capabilities include:
  - Dependency Injection
  - Persistence support (DAO and ORM)
  - AOP capabilities
  - Integration with standard Web technologies
  - An MVC based Web package

- Spring helps build complex enterprise applications
  - Provides a non-intrusive, lightweight solution for managing complex dependencies between Java objects
  - Supplies many of the important capabilities of Java EE in a less complex and lighter weight manner

Lesson Summary

- Spring supports **loose coupling** between components
  - Promotes technique of **coding to interfaces**, rather than concrete implementation classes
  - This by itself is an important design technique,
- Provides support for initialization of beans and their dependencies **through configuration**
  - Can be done using XML configuration files
- **Dependency Injection** is used by the Spring container to inject bean dependencies based on configuration data
  - Allows complete decoupling of your code from concrete implementation types

Module 2: More About Bean Properties

Working with Properties

Collection Valued Properties

Additional Capabilities

Lesson Objectives

- Gain a deeper understanding of Spring's core capabilities related to bean properties and dependencies
- Initialize simple and complex bean properties using Spring configuration
- Understand and use constructor injection, and understand its strengths and weaknesses
- Work with collection valued bean properties
- Understand how Spring supports factory methods
- Understand how autowiring works

Working with Properties

Working with Properties

Collection Valued Properties

Additional Capabilities

Configuring Value Based Properties

- Spring also allows configuration of **value based** properties
  - Containing simple values such as an *int* or *String*
- Similar to configuring dependencies
  - Configure the property value using a **string representation** of the bean property
  - At bottom, we configure a string property of *JavaInstructor* called *favoriteTopic*

```
public class JavaInstructor implements Teacher {
    // Much detail omitted …
    private String favoriteTopic;
    public void setFavoriteTopic (String topic) {
        favoriteTopic = topic;
   }
    public String getFavoriteTopic() { return favoriteTopic; }
}
```

```
<bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
  <property name="favoriteTopic" value="Dependency Injection"/>
  <!-- Other detail omitted … -->
</bean>
```

Using Value Based Properties

- The *favoriteTopic* property is set by the Spring container when the bean is instantiated
  - No dependency on any Spring API in *JavaInstructor*
- These properties are normal Java properties
  - For example, if the *Teacher* interface exposed the *favoriteTopic* property, we could use it as shown below
  - It's transparent to bean users that Spring is involved in initializing a property

```
ClassPathXmlApplicationContext ctx = new
        ClassPathXmlApplicationContext("applicationContext.xml");
Teacher teacher = ctx.getBean("springGuru", Teacher.class);
System.out.println("My favorite topic is " +
                        teacher.getFavoriteTopic());
```

Property Conversions

- Spring automatically converts strings in the configuration to the type of its property in the Java class
  - It uses Java *PropertyEditors*
  - Supports all the standard primitive and wrapper types
  - Below, a *java.lang.**Double*** property (called *age*) is configured
  - The string "*35*" is automatically converted to a *Double* value

```
public class JavaInstructor implements Teacher {
   // Much detail omitted …
   private Double age;
   public void setAge (Double d) {
       age = d;
  }
   public Double getAge() { return age; }
}
```

```
  <bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
    <property name="age" value="35"/>
    <!-- Other detail omitted … -->
  </bean>
```

Constructor Injection

- Properties can be injected using constructors
  - Via *<constructor-arg>* elements in the config file
  - Requires a matching constructor in the implementation class
  - Can initialize both simple properties and bean references

```
public class JavaInstructor implements Teacher {
   private InfoSource info;
   private String favoriteTopic;

   public JavaInstructor(InfoSource infoIn, String topicIn) {
      info = infoIn;
      favoriteTopic = topicIn;
   }
  // Much detail omitted …
}
```

```
   <bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
      <constructor-arg ref="springBook"/>
      <constructor-arg value="Dependency Injection"/>
      <property name="age" value="35"/>
   </bean>
```

Constructor Argument Resolution

- The container tries to resolve constructor elements in the configuration to constructor arguments in the bean definition
  - Can do this in multiple ways
- For example, the configuration shown below will work
  - Uses the *JavaInstructor* constructor seen previously
  - Note how the *springBook* reference is **last** in the configuration, but first in the constructor definition
  - Spring uses the constructor argument **types** first to determine what each *<constructor-arg>* element is for
  - The order is not needed in this case

```
<bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
    <constructor-arg value="Dependency Injection"/>
    <property name="age" value="35"/>
    <constructor-arg ref="springBook"/>
</bean>
```

Constructor Argument Resolution

- Consider the example at bottom though
  - The 2nd and 3rd constructor arguments can't be resolved by type
  - They are just strings in the config file
  - Here, the container will **use the order** in the config file to initialize the last two constructor arguments

```
public class JavaInstructor implements Teacher {
   private InfoSource info;
   private String favoriteTopic;
   private Double age;

   public JavaInstructor(InfoSource infoIn, String topic, Double ageIn) {
      info = infoIn;
      favoriteTopic = topic;
      age = ageIn;
   }
  // Much detail omitted …
}
```

```
   <bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
      <constructor-arg ref="springBook"/>
      <constructor-arg value="Dependency Injection"/>
      <constructor-arg value="35"/>
   </bean>
```

Constructor Argument Resolution

- Consider a new constructor definition:

  *public JavaInstructor(InfoSource infoIn, **Double ageIn, String topic**)*

  – The age argument is now before the topic string

  – The previous configuration entries will throw an exception (the container tries to convert "Dependency Injection" to a *Double*)

- **Specifying the type of the argument** can help argument resolution, as shown below

  – The container must be able to unambiguously resolve the arguments

  – You can also use indices *

```
<bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
    <constructor-arg ref="springBook"/>
    <constructor-arg type="java.lang.String" value="Dependency Injection"/>
    <constructor-arg type="java.lang.Double" value="35"/>
</bean>
```

XML Shortcuts - p: and c: Namespaces

- **p:** (**p-namespace**) allows using attributes to set property values
  - If our *Teacher* type had a "bestSubject" property, we could initialize it using the ***p:bestSubject*** syntax shown below
  - ***-ref*** after the property name would indicate the value is a reference

- **c:** (c-**namespace**) allows using attributes to set constructor args
  - If *JavaInstructor* had a constructor taking an InfoSource, we could initialize it with the ***c:infoIn-ref*** syntax shown at bottom
  - Here we use the -ref suffix to show that the value is a reference

```
<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor"
       p:bestSubject="Spring"/>
```

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

<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor"
       c:infoIn-ref="springBook"/>
```

Setter Injection vs. Constructor Injection

- Generally, setter injection is easier to use than constructor injection - however, they both have pros and cons

- **Setter Injection Pros**
  - **Much easier to use** - Multiple property definitions easier than a constructor with many arguments or multiple constructors
  - Can **flexibly choose** which properties to configure
  - Fairly easy to use with inheritance

- **Setter Injection Cons**
  - **Doesn't guarantee property initialization** (if you leave out a property configuration, it will simply remain unitialized)
  - **Requires setter methods** for any property you want to configure
  - **Doesn't support immutable beans** well, since you have to have setters for the properties

Setter Injection vs. Constructor Injection

- **Constructor Injection Pros**
  - **Guarantees properties are initialized** – if constructor argument is left out, you'll get an error when the bean is instantiated
  - **Setter methods not needed** for properties
  - **Immutable beans** easily supported (since setters not required)

- **Constructor Injection Cons**
  - **Constructors can be unwieldy** – Complex and hard to use with large numbers of properties
  - **Constructors offer less flexibility** – Separate constructors needed for different initialization scenarios
  - **More complex XML** for constructor injection than setter injection
    - Especially if type or index arguments are needed

Lab 2.1 – Spring Configuration

Lab 2.1 – Spring Configuration

Lab

- **Overview**: In this lab, we will use Spring in a slightly more complex manner
  - The lab is based on code from the JavaTunes online music store
  - You will configure a *Catalog* bean that provides a search service to search the JavaTunes catalog
  - *Catalog* uses an *ItemDAO* bean to do the actual search
  - You will be required to configure these beans, and dependencies between them, using constructor injection and setter injection
- **Objectives**:
  - Use the Spring configuration file and become more familiar with it
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

The JavaTunes Online Store

Lab

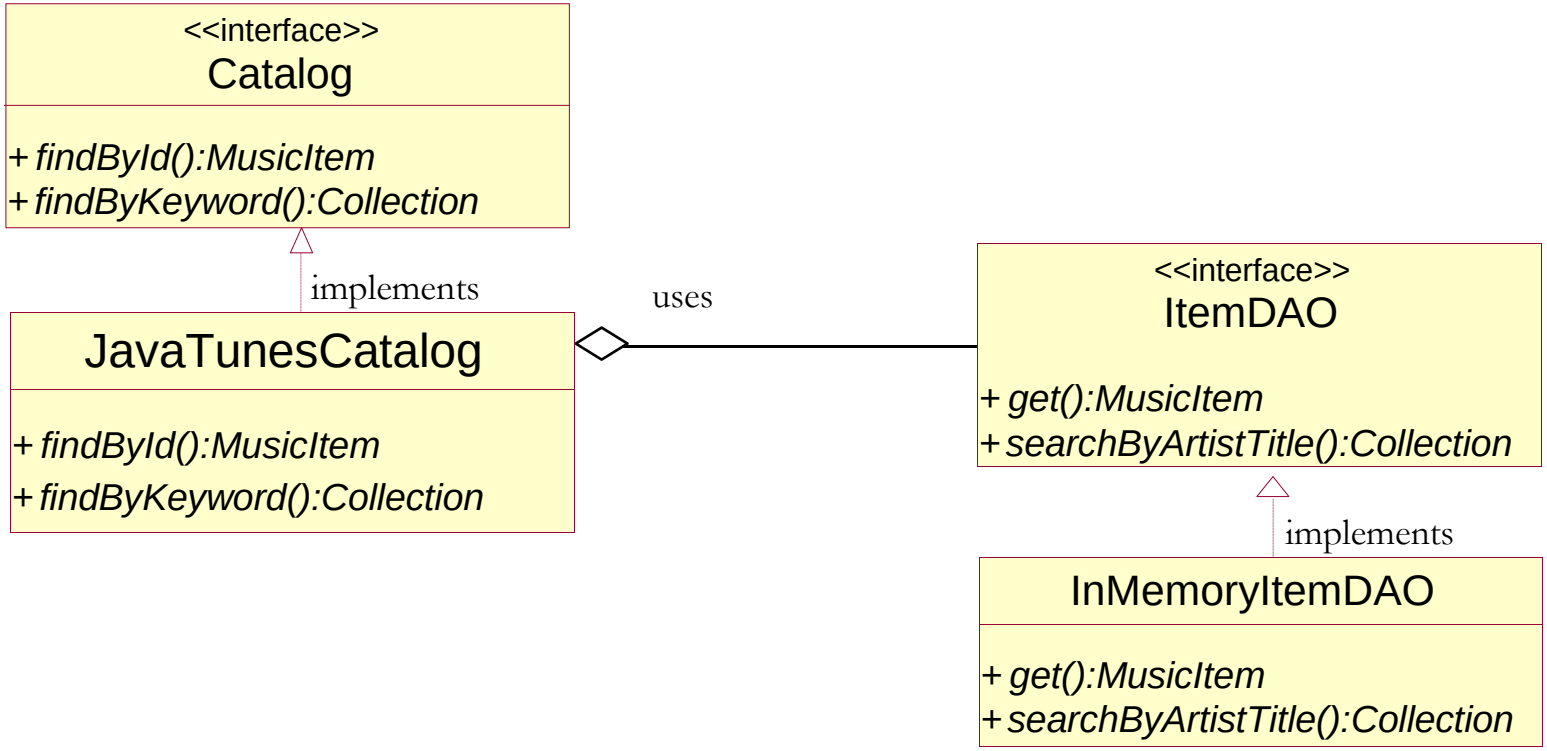
- The application we will be working with is a small piece of the JavaTunes online music store
  - We will be working with the business classes that perform a search, including the following classes:
  - ***com.javatunes.util.MusicItem*** : JavaBean-style value class that represents a music item (e.g. an mp3) for sale
  - ***com.javatunes.catalog.ItemDAO*** : Interface that defines data access for items
  - ***com.javatunes.catalog.InMemoryItemDAO*** : Concrete *ItemDAO* implementation (uses simple in-memory storage)
  - ***com.javatunes.catalog.Catalog*** : Interface that defines the functionality of a catalog in JavaTunes
  - ***com.javatunes.catalog.JavaTunesCatalog*** : Concrete *Catalog* implementation

- We **give you ALL** these classes – you don't need to write them

JavaTunes Class Diagram

Lab

- Shows the relationship between the JavaTunes classes
  - They use the DAO (Data Access Object) Pattern, which provides an abstract interface for data access (see notes)
  - You must wire the relationships up in the Spring configuration
  - There may be additional methods (not shown for simplicity)



Catalog and ItemDAO Interfaces

Lab

- We supply these interfaces - **You don't have to write them**
- They contain the methods shown below
  - *ItemDAO* and *Catalog* have very similar functionality
  - It's because we keep the example simple (see notes)

```
public interface ItemDAO {
  public MusicItem get(Long id);   // Find one MusicItem by id
  // Search for MusicItems containing keyword in title or artist
  public Collection<MusicItem> searchByArtistTitle(String keyword);
}
```

```
public interface Catalog {
    public MusicItem findById(Long id); // Find one MusicItem by id
    // Search for MusicItems containing keyword
    public Collection<MusicItem> findByKeyword(String keyword);
    // Other functionality to be added later …
}
```

JavaTunesCatalog Class

Lab

- We supply this class for you - **You don't have to write it**
- *JavaTunesCatalog* implements *Catalog*
  - It has a constructor and a property that we'll initialize using Spring DI

```
public class JavaTunesCatalog implements Catalog {

    private ItemDAO dao; // Field for dao

    private int maxSearchResults = 0; // maxSearchResults property
    public void setMaxSearchResults(int maxIn) {
        maxSearchResults = maxIn;
    }
    public int getMaxSearchResults() { return maxSearchResults; }

    // Constructor - We require a JavaTunesCatalog to be initialized
    // with an ItemDAO bean when it's created
    public JavaTunesCatalog(ItemDAO daoIn) {
        dao = daoIn;
    }
} // remaining detail omitted ...
```

Lab Preparation

Lab

- The new lab folder where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab02.1

Tasks to Perform

- Close all open files

- Create a new Java project called Lab02.1 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**

Implement the Interfaces

Lab

Tasks to Perform

- Open up *MusicItem* and review this class
  - It is a basic JavaBean with properties and very little else
  - There is nothing else you need to do with *MusicItem*
- Open up *InMemoryItemDAO*, and add code so *InMemoryItemDAO* implements *ItemDAO*
  - Review the class briefly also
  - The implementation is not that important – it uses a simple in-memory data store to emulate a database
- Open *JavaTunesCatalog*, and add code so *JavaTunesCatalog* implements *Catalog*
  - Review the class also, looking especially at the constructor and the *maxSearchResults* property
  - These will be initialized by injection using Spring
- Why are we using interfaces here?

Configure the Beans

Lab

Tasks to Perform

- Open *src/applicationContext.xml* and add: *
  - A *com.javatunes.catalog.InMemoryItemDAO* bean definition, with an id of *inMemoryItemDAO* (see notes about bean naming)
    - This requires no injection or other configuration
  - A *com.javatunes.catalog.JavaTunesCatalog* bean definition, with an id of *javaTunesCatalog*
    - This requires that you include a configuration element using constructor injection to pass in the needed *ItemDAO* bean (*inMemoryItemDAO*)
- Open up JavaTunes.java, look for the TODO comments, and add in the following code (see the previous lab code for help)
  - First, create an *ApplicationContext* that reads *applicationContext.xml* from the classpath
  - Next, look up the *javaTunesCatalog* bean from the context
  - Review the code that we've provided that calls the catalog's *findById* and *findByKeyword* methods and outputs the results

Run the Program

Lab

Tasks to Perform

- Make sure you have no compilation errors in your code
- **Run** the program in *JavaTunes.java* (as done previously *)
  - You should see output from the calls to the search methods we've included in the JavaTunes program – in particular, the call to *findByKeyword* on "a" should produce quite a few results
- Once this works, go back to *applicationContext.xml*, and configure a value for the ***maxSearchResults*** property of *javaTunesCatalog*
  - **Run** the JavaTunes program again
  - The output from the call to *findByKeyword* should now be limited to *maxSearchResults* items
- Congratulations – you've configured a complete app
  - You've used both constructor and setter injection

STOP

Collection Valued Properties

Working with Properties

Collection Valued Properties

Additional Capabilities

Working with Collections

- Spring can configure properties containing a collection
  - Four types of collection configuration elements:
  - **<list>**: A list of values, allowing duplicates
  - **<set>**: A set of values, allowing NO duplicates
  - **<map>**: A set of name-value pairs, where name and value can be of any type
  - **<props>**: A set of name-value pairs, where the name and value are both *Strings*

- <*list*> and <*set*> can be used for **array** valued properties or for implementations of *java.util.Collection*
  - The actual bean property does not have to be *List* or *Set*
  - The container enforces the list or set contract

Collection Property Example

- Consider a collection valued property, as shown below
  - The *favoriteTopics* property is now defined as a collection
  - A teacher can now have multiple favorite topics that are contained in this collection property
  - We'll see how to configure this next

```
public interface Teacher {
   // Much detail omitted …
   public Collection<String> getFavoriteTopics();
}
```

```
public class JavaInstructor implements Teacher {
   // Much detail omitted …
   private Collection<String> favoriteTopics;
   public void setFavoriteTopics (Collection<String> topics) {
       favoriteTopics = topics;
  }
   public Collection<String> getFavoriteTopics() { return favoriteTopics; }
}
```

Configuring <list> and <set> Properties

- We can initialize the *favoriteTopics* property using either of the examples shown below
  - The *<list>* element will result in a collection with 3 values
  - The *<set>* element will filter out the duplicate "*Spring*" value, and will result in a collection with 2 values

```
<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor">
  <property name="favoriteTopics">
    <list>
      <value>Spring</value>
      <value>Java</value>
      <value>Spring</value>
    </list>
  </property>
</bean>
```

```
<property name="favoriteTopics"> <!-- Most details omitted -->
  <set>
    <value>Spring</value>
    <value>Java</value>
    <value>Spring</value>
  </set>
</property>
```

Configuring Collection of Bean References

- Collection properties may also contain references to other beans (dependencies)
  - They are configured similarly to value based collections
  - For example, if *JavaInstructor* had the following Constructor:
    JavaInstructor(Set<InfoSource> infoSources)
  - You could configure it using the configuration elements below

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

<bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
    <constructor-arg>
        <set>
            <ref bean="springBook"/>
            <ref bean="javaBook"/>
        </set>
    </constructor-arg>
</bean>
```

Map Valued Properties

- *Map* valued properties can also be configured with Spring
  - For example, if *JavaInstructor* had the following property: ***Map<String,InfoSource> infoSources;***
  - You could configure it using the configuration elements below

```
<bean id="springBook" class="com.javatunes.teach.SpringCourseBook"/>
<bean id="javaBook" class="com.javatunes.teach.JavaCourseBook"/>
<bean id="totalGuru" class="com.javatunes.teach.JavaInstructor">
  <property name="infoSources">
    <map>
      <entry>
        <key><value>Spring Book</value></key>
        <ref bean="springBook"/>
      </entry>
      <entry>
        <key><value>Java Book</value></key>
        <ref bean="javaBook"/>
      </entry>
    </map>
  </property>
</bean>
```

java.util.Properties Valued Properties

- *java.util.**Properties*** valued properties can also used
  - A *Properties* object holds *String* names and values
  - For example, if *JavaInstructor* had the following property: ***Properties topicExpertise;***
  - Assume this property holds a topic as the key, and a description of the Instructor's expertise as a value
  - You could configure it using the configuration elements below

```
<bean id="totalGuru" class="com.javatunes.teach.JavaInstructor">
  <property name="topicExpertise">
    <props>
      <prop key="Spring">Expert</prop>
      <prop key="Java">Expert</prop>
      <prop key="EJB3">Good</prop>
    </props>
  </property>
</bean>
```

Lab 2.2 – Collection Valued Properties

Lab 2.2 – Configuring Collections

Lab

- **Overview**: In this lab, we will initialize a collection valued property by injection
  - The lab is also based on JavaTunes code
  - It modifies the *InMemoryItemDAO* class so its in-memory storage is initialized by injection, rather than hard coded
- **Objectives**:
  - Use the Spring configuration file to inject a collection valued property
- **Builds on previous labs**: none
- **Approximate Time**: 20-30 minutes

Lab Preparation

Lab

- The new lab folder where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab02.2

Tasks to Perform

- Close all open files

- Create a new Java project called Lab02.2 in the workspace *
  - See Lab 1.1 instructions on how to do this
  - Remember to **add the Spring user library**

Lab Preparation

Lab

Tasks to Perform

- **Open** *InMemoryItemDAO.java* for review
  - **Note that it now has a constructor that receives a collection** representing its in-memory store
  - **You'll need to supply this collection via constructor injection** in the Spring configuration file
  - There is **nothing you need to code** in this class
- **Open** *src/applicationContext.xml*
  - Note the configuration of numerous *MusicItem* beans, with ids of *mi1*, *mi2*, and etc.
  - Go to the *inMemoryItemDAO* bean definition, and add in a constructor argument containing a list of *MusicItem* references
  - Use some of the *MusicItem* beans configured later in the file as the contents of the list (make sure to include the bean with id = mi2, since we search on it in the test program)

Configure the Beans

Lab

Tasks to Perform

- **Run** the program in JavaTunes.java as before *
- If you've configured the constructor argument correctly, your program will run without any exceptions
  - You should see output from the calls to the search methods we've included in the JavaTunes program
- Congratulations – you've configured a complete app
  - You've used both constructor and setter injection

STOP

Additional Capabilities

Working with Properties

Collection Valued Properties

Additional Capabilities

- Factory Methods -

- Spring supports the creation of beans via factory methods
  - First, we consider **static factory methods**
- The ***getInstance()*** method, shown below, is used to create *JavaInstructor* instances
  - Rather than using *new JavaInstructor()*
- The ***<factory-method>*** attribute of *<bean>* tells Spring to use a factory method, as shown at bottom

```
public class JavaInstructor implements Teacher { // Details omitted
   public static JavaInstructor getInstance() { /* ... */ }
}
```

```
<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor"
       factory-method="getInstance">
  <!-- Most detail omitted -->
</bean>
```

Instance Factory Methods

- Spring also supports **instance factory methods**
  - Often encountered when using factory classes
- ***JavaInstructorFactory***, shown below, has an instance method to produce *JavaInstructor*s
- *<bean>* uses a ***factory-bean*** attribute rather than *class*
  - The factory method is invoked on the specified factory bean **instance**

```
public class JavaInstructorFactory { // Details omitted
    public JavaInstructor getInstance() { /* ... */ }
}
```

```
<bean id="myFactory"
        class=com.javatunes.teach.JavaInstructorFactory"/>

<bean  id="springGuru"
        factory-bean="myFactory"   factory-method="getInstance">
</bean>
```

- Bean Aliases -

- A bean must have at lease one id, but may have more
  - Ids can be specified with the ***id*** or ***name*** attributes
- ***id*** attribute: Lets you specify exactly one id, must follow rules for XML ids - e.g. it can't start with a number
- ***name*** attribute: Can hold multiple values, doesn't have to follow rules for XML ids
  - Multiple values can be separated by a comma (,) semicolon (;) or white space
  - Below, we give a bean 2 names that are not valid XML ids

```
<bean name="1-springBook, 2-javaBook"
      class="com.javatunes.teach.SpringCourseBook"/>

<bean id="springGuru" class="com.javatunes.teach.JavaInstructor">
   <property name="info" ref="1-springBook"/>
</bean>
```

- Bean Definition Inheritance -

- A bean definition (the **child**) can **inherit** configuration data from another bean (the **parent**)
  - Useful when configuration data is common to multiple beans
  - Possible to override some values, or add values

- The ***parent*** attribute is used to specify that a bean inherits configuration data from another bean
  - Inherits the bean class, constructor-args, and all the properties
  - Can override all of these
  - Certain settings always taken from the child *

- An ***abstract="true"*** attribute is available for parent beans
  - The container won't instantiate abstract beans
  - Useful if you're defining the bean ONLY to share configuration

Inheritance Example

- The example below shows both inheritance and overriding

```
<bean id="seniorJavaInstructor" class="com.javatunes.teach.JavaInstructor">
 <property name="infoSources">
      <set>
          <ref bean="springBook"/>
          <ref bean="javaBook"/>
      </set>
   </property>
  <property name="favoriteTopics">
    <list>
      <value>Spring</value>
      <value>Java</value>
      <value>Hibernate</value>
    </list>
  </property>
</bean>
                          <!- Inherit definitions from parent -->
<bean id="springGuru" parent="seniorJavaInstructor">
  <property name="favoriteTopics">  <!-- Override this property -->
    <set>
      <value>Spring</value>
    </set>
  </property>
  <property name="age" value="35"/>
</bean>
```

- SpEL: Spring Expression Language -

- **Spring Expression Language** (**SpEL**) introduced in Spring 3
  - Allows wiring values into beans, including:
    - Values from other bean properties
    - Method return values, including static methods of a non-bean class
    - System properties
- A SpEL expression has the form ***#{expression string}***
  - For example, if our springGuru (Teacher) bean had a "bestSubject" property, we could access it as follows:

    #{springGuru.bestSubject}

    - *springGuru* is a bean id and *bestSubject* is a property of the bean
  - When used in a config file, it would look like the example below
  - When used with annotations, it would like the example at bottom

```
<property name="subject" value="#{springGuru.bestSubject}" />
```

```
@Value( "#{springGuru.bestSubject}" )
private String bestSubject;
```

Other SpEL Capabilities

- Invoke methods on a bean
  - For example, to invoke *pickASubject()* on *springGuru*

```
#{springGuru.pickASubject()}
```

- Access system properties through the predefined *systemProperties* variable
  - For example, to get the *todaysSubject* system property

```
#{systemProperties.todaysSubject}
```

- Access static members of a class using the T (type) operator
  - For example, you can call *java.lang.Math.random()* as follows

```
#{ T(java.lang.Math).random()  }
```

- Can also be used to evaluate expressions in Java programs through the use of *ExpressionParser* and other classes
- See the reference manual for the wide range of SpEL's capabilities

- Autowiring -

- Spring can automatically resolve bean dependencies
  - Called **autowiring**, and can be done four different ways
- **byName** : Attempts to find a bean named exactly the same as the property which needs to be autowired
- **byType** : Attempts to find a single bean whose type matches the type of the property being wired
  - If there is more than one match, a fatal exception is thrown.
  - If there are no matching beans the property is not set.
- **constructor** : Analogous to byType, but applies to constructor args
- **autodetect** : Attempts to wire *constructor* and then *byType* through introspection of the bean class
  - If a default constructor is found, *byType* mode will be applied

Autowiring byName Example

- Autowiring specified via ***autowire*** attribute of *<bean>*
  - The default is *autowire="no"* which specifies no autowiring
  - You specify one of the other four possibilities to trigger autowiring
- Below, the *JavaInstructor* ***info*** property is autowired to the bean named ***info*** (a *SpringCourseBook*)
  - This uses *byName* autowiring

```
public class SpringCourseBook implements InfoSource { /* … */ }
```

```
public class JavaInstructor implements Teacher { // Detail omitted
   private InfoSource info;
   public void setInfo (InfoSource infoIn) {
       info=infoIn;
  }
}
```

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

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

Autowiring byType

- The first example below shows a successful use of *byType*
  - There is one bean of type *InfoSource*, and so it is wired to the *JavaInstructor info* property
- The second example below will fail
  - There are two beans of type *InfoSource*
  - Assumes that *JavaCourseBook* also implements *InfoSource*

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

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

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

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

Pros and Cons of Autowiring

- Autowiring: Saves you the trouble of wiring beans **with XML**
  - Can make your configuration files simpler
  - Can help your configuration files evolve "automatically"
  - Very useful in the common case where there is only one instance of a particular bean type
- Some **disadvantages** to autowiring
  - **Less clear** than explicit wiring
  - **You don't see the exact configuration**, since it's done by the container - creating a place for errors to enter
  - *byName* and *byType* place **requirements and restrictions** on the names or types of beans you can have in a configuration
  - **Can lead to unforeseen results** when you make modifications
    - e.g. if you change the name of a property wired *byName*, behind the scenes it may get wired to something you didn't expect

To Autowire or Not to Autowire

- Autowiring lacks clarity, and may change in unexpected ways as you maintain and evolve your program
  - Tools also do not support autowiring very well

- Because of this, autowiring should be used carefully
  - Very useful in situations where it's not likely to give an error
    - e.g. - where there is only one bean instance of a given type
  - Perhaps avoid it with complex systems where the autowiring may inadvertently change, or where the complexity is confusing

- It is available in the Spring framework, and should be evaluated carefully
  - To make sure it is not likely to introduce subtle bugs

Review Questions

- What capabilities does Spring give you for initializing simple value properties?
- What is constructor injection, and how does it compare to setter injection?
- What capabilities does Spring give you for initializing collection valued properties?
- How does Spring support factory methods?
- What is autowiring?

Lesson Summary

- Spring has simple configuration elements to configure and initialize **value based** properties
  - Spring can automatically convert string values in the configuration into the proper type of a property
    - Using JavaBean *PropertyEditors*
- Spring can also inject properties using **constructors**
  - Uses <*constructor-arg*> elements in the configuration file
  - Requires a constructor be defined in the Java class
  - Container automatically tries to resolve constructor arguments in the config file to the proper arguments in the Java constructor
  - Constructor injection is useful to guarantee that a property is initialized,
  - Setter injection is generally easier and more flexible to use

Lesson Summary

- Spring supports *list*, *set*, *map* and *props* for *collection valued properties*
  - Can be configured in the Spring config file
  - Can hold simple values, bean references, other collections ...
- Spring supports the use of both **static and instance factory methods** to create bean instances
  - Uses additional bean attributes to specify the factory method and, if needed, the factory bean
- With **autowiring**, the Spring container can resolve bean dependencies
  - Can be done *byType*, *byName*, *constructor*, or *autodetect*
  - Ambiguity may be introduced when using autowiring, and should be evaluated carefully

Module 3: The Spring Container and API

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

Lesson Objectives

- Learn more about the **Spring container**
  - Details of the *ApplicationContext* API
  - Be familiar with the Spring *Resource* classes
- Understand the **lifecycle and scope** of managed beans
  - Configure singleton and prototype beans
  - Work with Spring bean lifecycle events
- Learn how to use **Spring annotations** for configuration

*

ApplicationContext

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

ApplicationContext Interface

- The *org.springframework.context.***ApplicationContext** interface extends *BeanFactory*,

- Exposes large number of Spring's capabilities, including:
  - **Accessing Resources** like URLS & files - e.g. config files
  - **Message resources with I18N** capability
  - **Publishing events** to managed beans that are listeners
  - **Multiple, hierarchical contexts**

- Generally you'll use *ApplicationContext* rather than *BeanFactory*
  - Builds on *BeanFactory*, and adds additional capabilities
  - Recommended practice is to use *ApplicationContext*
  - Many capabilities not supported by *BeanFactory*

ApplicationContext Implementations

- Two *ApplicationContext* implementations that are commonly used directly,
  - In the *org.springframework.context.support* package
  - ***ClassPathXmlApplicationContext***: Standalone context that loads XML context definition files from the classpath
  - ***FileSystemXmlApplicationContext***: Standalone context that loads XML context definition files from the file system or URLs
- *ApplicationContext* extends *BeanFactory* and inherits all its capability
  - For example, you retrieve a bean using *getBean()*

Constructors

- *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* have a number of similar constructors, shown below
  - They are listed by their arguments:
  - **(String… configLocation)**: Load the configuration from the given XML file(s) and automatically refresh the context (see notes)
  - **(String[] configLocations)**: Load the configuration from the given XML files and automatically refresh the context
  - **(String[] configLocations, ApplicationContext parent)**: Create a new context with the given parent, loading the configuration from the given XML files and automatically refreshing the context.
  - **(String[] configLocations, boolean refresh)**: Load the configuration from the given XML files
  - **(String[] configLocations, boolean refresh, ApplicationContext parent)**: Create a new context with the given parent, loading the configuration from the given XML files

Using an ApplicationContext

- Below, ***ClassPathXmlApplicationContext*** is used to load a definitions file
  - *applicationContext.xml* **must be in the classpath**
  - It's very easy to load multiple files – as shown at bottom
  - You can also use <*import*> in the XML file to import definitions from other XML config files (see notes)

```
package com.javatunes.teach;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TeachMeSpring {
   public static void main(String[] args) {
         ApplicationContext ctx =
               new ClassPathXmlApplicationContext("applicationContext.xml");
      Teacher teacher = ctx.getBean("springGuru", Teacher.class);
      teacher.teach();    // Use our bean
   }
}
```

```
// Load configuration from two files
ApplicationContext ctx =
     new ClassPathXmlApplicationContext( "ctx-1.xml", "ctx-2.xml" );
```

Spring Resource Access

- Spring uses its ***Resource*** classes under the hood to access the config info for *ClassPathXmlApplicationContext*
  - They provide extensive functionality for accessing resources
    - e.g. files on the file system
- ant style wildcards can be used to specify config files
  - **conf/**/ctx.xml**: All *ctx.xml* files under any subdir of conf
  - **conf/*-context.xml**: All files in conf, ending in –context.xml
- Can also use the **file:** and **classpath:** prefixes
  - Forces use of the specified loading mechanism
    - Independently of the application context implementation
  - e.g. The following loads definitions from the classpath
    - Even though *FileSystemXmlApplicationContext* is used

```
new FileSystemXmlApplicationContext("classpath:ctx.xml");
```

Built-in Resource Implementations

- Some important *Resource* implementations are listed below
  - All in the *org.springframework.core.io* package:
  - ***ByteArrayResource***: Implementation for accessing a byte array
  - ***ClassPathResource***: Represents a resource which should be obtained from the classpath
  - ***FileSystemResource***: Resource implementation for *java.io.File* handles - Supports resolution as *File*, and *URL*
  - ***UrlResource***: Resource implementation for *java.net.URL* locators. Supports resolution as *URL*, and also as *File* for the "*file:*" protocol

- They provide a standard way to access resources
  - Often wrap an underlying Java class (such as URL) and add additional functionality for ease of use
  - Resource classes are not used that much directly
  - Application context implementations already include this functionality

Lab 3.1 – More about ApplicationContexts

Lab 3.1 – ApplicationContext

Lab

- **Overview**: In this lab, we will work with more of the *ApplicationContext* API
  - We'll use *ClassPathXmlApplicationContext* to load configuration files from the classpath, and to split our configuration into multiple files
- **Objectives**:
  - Use the Spring *ApplicationContext*
- **Builds on previous labs**: 2.2
  - Continue to work in the **Lab02.2** project
- **Approximate Time**: 25-35 minutes

Multiple Configuration Files

Lab

Tasks to Perform

- We will first split up the spring configuration into two files
  - One will contain the DAO bean and the individual item beans
  - The other will contain the catalog bean
- The easiest way to do this is to copy *applicationContext.xml* to a file *inMemoryItemDAO.xml*, then do the following
  - In *applicationContext.xml*, delete all the *MusicItem* bean definitions, and the *inMemoryItemDAO* bean definition
    - Leaving only the *javaTunesCatalog* def
  - In *inMemoryItemDAO.xml* delete the *javaTunesCatalog* def
    - Leaving the *inMemoryItemDAO* and *MusicItem* defs
- In *JavaTunes.java,* pass the two bean definition file names to the *ApplicationContext* constructor

Multiple Configuration Files

Lab

Tasks to Perform

- **Run** your program – it should run as before
- Once your program is running with two config files, then try accessing them in different ways
  - For example, you can use wildcards, or try the file: prefix
    - See notes about wildcards
  - Note that the file: prefix will search for files relative to the root folder of the project
    - Because the program is run in that folder
- **[Optional]**: Look at the javadocs for *ApplicationContext*
  - Try some of the other methods - for example, you can get a bean count and bean names from the methods defined in *ListableBeanFactory*, which *ApplicationContext* implements

STOP

Bean Scope and Lifecycle

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

Bean Scope

- By default, all Spring configured beans are **singletons**
  - Only one instance created by container
  - Used whenever that bean is needed – whether for dependency injection, or for *getBean()*
- Bean **scope** can configure different behavior
  - Via the *scope* attribute of *<bean>*
- Available Spring scopes:
  - **singleton**: One bean definition produces a **single object instance** per Spring container
    - All requests for that bean return that instance
    - This is the default scope
  - **prototype**: One bean definition produces **multiple object instances**
    - **New instance created** every time a that bean is needed
  - **request, Module**
    , global Module
    Only valid in the context of a web- aware Spring ApplicationContext,

Bean Scope

and not covered now

Specifying Bean Scope

- The *springGuru* bean is scoped as ***prototype*** in the example below
  - A new instance is created every time *springGuru* is used
  - The two *springGuru* references in the example cause the container to instantiate **two instances** of *JavaInstructor*

```
<!-- Much detail omitted -->
<bean id="springGuru" class="com.javatunes.teach.JavaInstructor"
      scope="prototype">
</bean>

<bean id="someCourse" class="com.javatunes.teach.JavaCourse">
   <property name="teacher" ref="springGuru"/>
</bean>
<bean id="anotherCourse" class="com.javatunes.teach.JavaCourse">
   <property name="teacher" ref="springGuru"/>
</bean>
```

Inner Beans

- An **inner bean** is defined inside a *<property>* or *<constructor-arg>* element
  - Anonymous and don't need an id
  - Not referred to by name, but accessed through the property
  - Always scoped as prototype (no prototype attribute needed)
  - The containing bean owns the inner bean completely

- *springGuru* below, uses an inner bean for its *info* property
  - The info bean has no id declared
  - Only accessed through *JavaInstructor.getInfo()*

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

Compound Names

- **Compound property names** can be used to access nested beans
  - Consider a *TeacherData* type with a *name* (*String*), and *age* (*Double*) properties, and *JavaInstructor* as defined below
  - At bottom, we use ***data.age*** to access the *age* property within the *data* bean

```
public class JavaInstructor {
   private TeacherData data;
   public void setData(TeacherData dataIn) {
      data = dataIn;
   }
   public TeacherData getData() { return data; }
}
```

```
<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor">
   <property name="data">
      <bean   class="com.javatunes.teach.TeacherData"/>
   </property>
   <property name="data.age" value="35"/>
</bean>
```

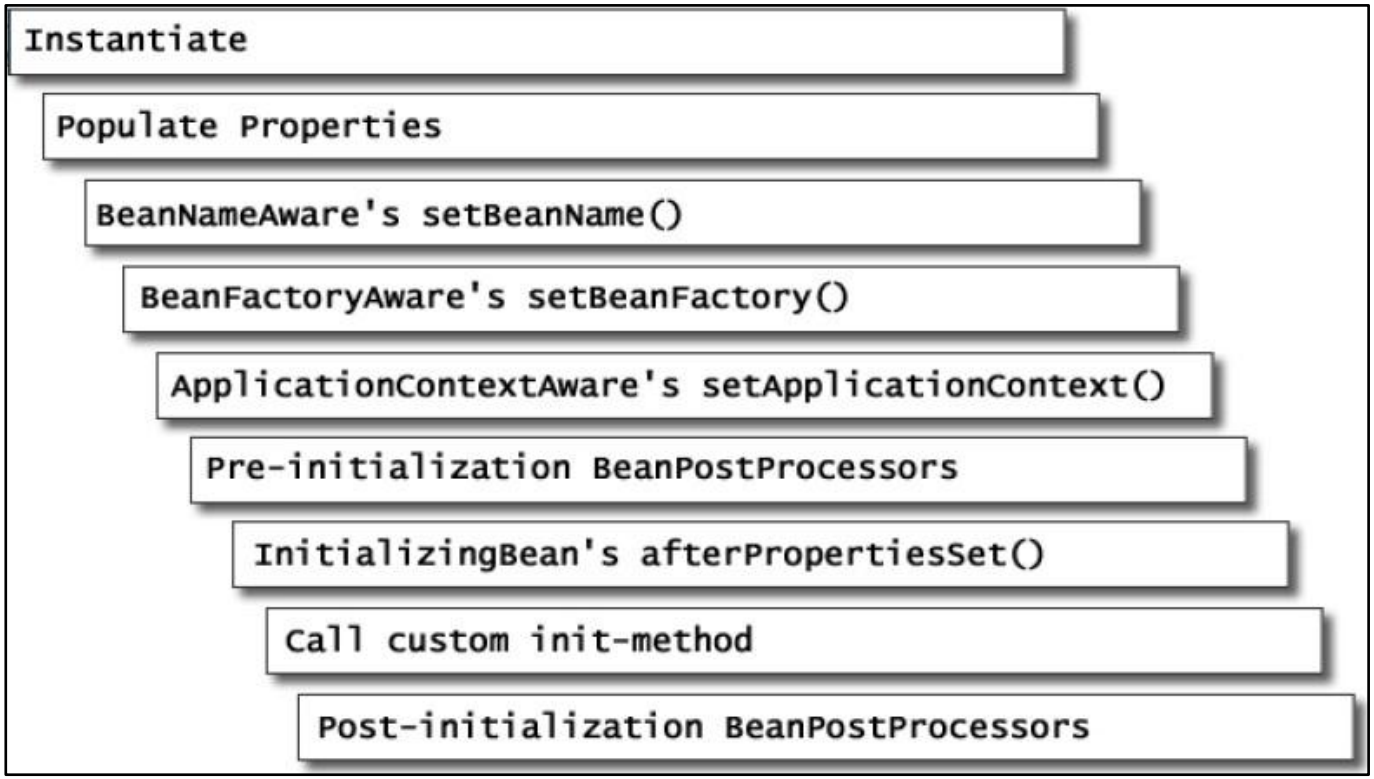
Depends On

- Dependencies are usually expressed by having one bean injected into the property of another
  - So the container naturally knows about the dependency
- Sometimes there are **less direct dependencies**
  - For example, static initializers that must be triggered
    - Like a database driver registration
- A bean with a ***depends-on*** attribute forces initialization of the other bean(s) before its own initialization
  - Below, *instructorDAO* will be created first

```
<bean  id="springGuru"  class="com.javatunes.teach.JavaInstructor"
       depends-on="instructorDAO">
</bean>

<bean id="instructorDAO" class="com.javatunes.teach.InstructorDAO"/>
```

Bean Creation Lifecycle

- Beans have a fairly sophisticated lifecycle within a Spring container
  - You can interact with parts of this lifecycle
  - Below, we show the steps in a typical bean's lifecycle when created
  - Spring interfaces are used to participate in the lifecycle



Bean Creation Lifecycle Details

- **Instantiation**: Container creates the bean instance
- **Populate properties**: Container injects bean's properties
- **Set bean name**: If bean implements *BeanNameAware*, call *setBeanName()* passing in bean's name
- **Set bean factory**: If bean implements *BeanFactoryAware*, call *setBeanFactory()* passing in the bean factory
- **Set application context**: If bean implements *ApplicationContextAware*, call *setApplicationContext()*
- **Postprocess**: If there are any *BeanPostProcessor*s, call their *postProcessBeforeInitialization()* methods
- **Initialize bean**: If bean implements *InitializingBean* , call *afterPropertiesSet()*. If bean has a custom init method, call it
- **Postprocess**: If there are any *BeanPostProcessors*, call their *postProcessAfterInitialization()* methods
- Bean is ready to use !

Using the Lifecycle Interfaces for Beans

- All lifecycle interfaces follow the same structure
  - For a bean implementing a lifecycle interface, its method is called at the appropriate time in the lifecycle
  - Below is an example using *ApplicationContextAware*
  - *setApplicationConext()* will be called with the owning context of the bean instance passed in
  - The bean instance can save it to do things – for instance look things up in the context

```
// Most detail omitted …
import org.springframework.context.ApplicationContextAware;

public class JavaInstructor implements Teacher,
                                        ApplicationContextAware {
   ApplicationContext ctx;
   void setApplicationContext(ApplicationContext ctxIn) {
     ctx = ctxIn;
   }
}
```

Bean Destruction Lifecycle

- One callback, *DisposableBean.destroy()*, for bean destruction
  - If bean implements *DisposableBean*, *destroy()* is called upon instance destruction
  - e.g. When the container shuts down
- Beans can have a custom destroy method for the same purpose
  - Called when bean is destroyed
  - Declared using the *destroy-method* attribute in the *<bean>* element
  - Below, *JavaInstructor* defines a destroy method called *cleanup*
- The example below also defines definition init method called *init*
  - The int method is called when the instance is created

```
<!-- Much detail omitted -->
<bean id="springGuru" class="com.javatunes.teach.JavaInstructor"
        scope="prototype"
        init-method="init" destroy-method="cleanup">
```

BeanPostProcessor

- You can define post processors involved in a beans lifecycle
- Done by defining a class implementing ***BeanPostProcessor***
  - This interface defines two methods:

```
Object postProcessBeforeInitialization(
                    Object bean, String beanName)
Object postProcessAfterInitialization(
                    Object bean, String beanName)
```

- Configuring a bean implementing *BeanPostProcessor* triggers calls on its methods for **every** bean created by the container
  - At the postprocess point in the lifecycle shown earlier
  - You can define as many *BeanPostProcessor*s as you want

@PostConstruct and @PreDestroy

- You can also use standard Java lifecycle annotations
- *@PostConstruct* annotated methods called where *InitializingBean.afterPropertiesSet()* would be
- *@PreDestroy* annotated methods called where *DisposableBean.destroy()* would be
- Can mix and match these mechanisms
  - Can also use both, e.g. define *afterPropertiesSet()* , and annotate a method with *@PostConstruct*
  - Both will be called, with the *@PostConstruct* method called before *afterPropertiesSet()*
- Annotations are convenient, but the interfaces have more capability

Event Handling

- *ApplicationContext* can notify beans of application events
  - Beans receive events if they implement *ApplicationListener*
  - The interface is parameterized, so only events of the appropriate type are received (*ApplicationEvent* in the example below)
- The container itself publishes events (see notes)
  - User beans can also publish custom events by calling *publishEvent()* on the *ApplicationContext*

```
public interface ApplicationListener<E extends ApplicationEvent> {
  void onApplicationEvent(E event); // Handle an event
}
```

```
public class MyListener
                    implements ApplicationListener<ApplicationEvent> {
  void onApplicationEvent(ApplicationEvent event) {
    System.out.println(event);
  }
}
```

Lab 3.2 – Bean Lifecycle

Lab 3.2 – Bean Lifecycle

Lab

- **Overview**: In this lab, we will work with the bean lifecycle
  - We will declare some of our beans as inner beans
  - We will work with some of the lifecycle callbacks Spring provides
- **Objectives**:
  - Use inner beans
  - Become more familiar with the bean lifecycle and the interfaces Spring provides to work with them
- **Builds on previous labs**: 3.1
- **Approximate Time**: 25-45 minutes (depending on optional parts)

Using Inner Beans

Lab

Tasks to Perform

- Continue to work in the **Lab02.2** directory
- Open up the *inMemoryItemDAO.xml* file
  - Replace the bean references passed to the *InMemoryItemDAO* class with inner beans as follows
  - In the *<constructor-arg>* for the *inMemoryItemDAO* bean, delete all the *<ref>* elements within the *<list>* element
  - Move all the *<bean>* elements for the *MusicItem* beans from the top level of the file into the *<list>* element
- **Run your program again** – it should work as before
  - In *JavaTunes.java*, try to look up one of these inner beans by id (if you look, they all still have ids – mi1, mi2, …)
  - The lookup should fail, since the container won't assign ids to the inner beans, even if you include them in the configuration
  - **Comment out** or delete this lookup once you've seen the exception

Bean Lifecycle Callbacks

Lab

Tasks to Perform

- **Open *JavaTunesCatalog.java*** for editing
- **Modify** *JavaTunesCatalog* so it implements *InitializingBean*
  - This is in the *org.springframework.beans.factory* package
  - You'll need to add the following method from this interface:
    public void afterPropertiesSet()
    - Just print out a message so we can see it was called
  - Add a similar print statement to *setMaxSearchResults()*, so we can see when it is called relative to *afterPropertiesSet()*
- **Add** the following method to *JavaTunesCatalog*:
  public void init() – Just have it print out a message also
- **Edit *applicationContext.xml***, and add in an *init-method* attribute to *javaTunesCatalog*, specifying the *init()* method
- **Run your application again** – you should be able to see the messages from the lifecycle methods

[Optional] Bean Lifecycle Callbacks

Lab

[Optional] Tasks to Perform

- You can change your logging level to warn if you want to see the callback output more easily (see notes)
- If you have time, try working with some of the other lifecycle callback interfaces – for example, you can try:
  - Working with the destroy bean callbacks
  - Adding in global post processors by defining a class that implements *BeanPostProcessor*, and declaring a bean of that class in the *applicationContext.xml* file
  - All the callback methods you define should print out a message so you can see that it has been called
- Look at the other lifecycle callbacks, and try any that seem interesting to you

[Optional] Application Event

Lab

[Optional] Tasks to Perform

- If you have time, have *JavaTunesCatalog* implement the *ApplicationListener<ApplicationEvent>* interface
  - Define ***onApplicationEvent****(ApplicationEvent event)*
  - Have this method get the classname of the event (*event.getClass().getName()*) and print it out
  - You'll see that the container publishes some events
- Call *close()* on the context (in *JavaTunes.java* at the very end of the program)
  - You should see an additional event being called

STOP

Annotation Driven Configuration

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

Annotations in Spring

- **Annotations for configuration** introduced in Spring 2.x +
  - Usable in place of, or in addition to, XML configuration
  - We'll show an example, then explain them
- ***@Required*** on a setter indicates the property must be set in the configuration
  - Otherwise, the container throws an exception
  - Below, we specify ***info* MUST be initialized by dependency injection** in the configuration

```
import   org.springframework.beans.factory.annotation.Required;
public class JavaInstructor implements Teacher { // Detail omitted
   private InfoSource info;

   @Required
   public void setInfo (InfoSource infoIn) {
      info=infoIn;
   }
}
```

Enabling Spring Annotations

- *@Required* is detected by a bean post-processor
  - ***RequiredAnnotationBeanPostProcessor*** must be configured as a bean to use *@Required* (see below)
- Other annotations exist, and require other post-processors *
  - Can all be configured via ***context:annotation-config*** (see bottom)
  - You can configure individual beans, but that's usually not done

```
<bean class=
"org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"
/> <!-- Individual post-processor configuration example -->
```

```
<?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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">

      <context:annotation-config/>
</beans> <!-- tag configuration example -->
```

@Component and Auto-Detecting Beans

- ***@Component***: Indicates a class is a component
  - i.e. a Spring managed bean
  - In package *org.springframework.stereotype*
  - Called a "stereotype" annotation
- Spring can scan for and **auto-detect** annotated beans on the classpath
  - Which are registered as normal beans in the application context
  - XML definition isn't necessary
- ***<context:component-scan>*** in the XML configuration enables auto scanning
  - Also enables all capabilities of *context:annotation-config*
  - ***basePackage*** attribute lists packages to include in the scan

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

Annotation-based Bean Definition and DI

- ***@Resource*** can specify dependency injection
  - Usable on a field or on a setter method
  - It's a standard annotation in ***javax.annotation*** (in Java 6+ JDK)
  - The *name* element usable to specify name of the bean to be injected
  - Can also inject well-known resources like the application context
- Below, *JavaInstructor* is declared as a bean with ***@Component***
  - We inject the app context and *springBook* bean via ***@Resource***

```
import  javax.annotation.Resource;
import org.springframework.stereotype.Component;

@Component("springGuru") // Specify id (default is class name )
public class JavaInstructor implements Teacher { // Detail omitted
   private InfoSource info;
   @Resource // Inject application context
   private ApplicationContext ctx;

   @Resource(name="springBook") // Assumes springBook bean exists
   public void setInfo (InfoSource infoIn) { info=infoIn; }
}
```

Complete Declarations Using Annotations

- Below, we show two annotation-based bean definitions

```
package com.javatunes.teach;
import org.springframework.stereotype.Component;

@Component("springBook") // Declare as bean with name springBook
public class SpringCourseBook implements InfoSource {
   public String getData() {
      return "Dependencies are not so cool";
   }
}
```

```
package com.javatunes.teach;
import org.springframework.stereotype.Component;
import javax.annotation.Resource;

@Component("springGuru")
public class JavaInstructor implements Teacher { // Detail omitted
   @Resource(name="springBook") // Inject springBook bean
   private InfoSource info; // Don't even need a setter
}
```

XML Config for Annotations and Scanning

- Below, is the Spring configuration for the previous bean examples
  - The ***component-scan*** element configures the auto-scanning
    - It is part of Spring's *context* namespace
    - The ***base-package*** attribute declares that the auto-scanning should include classes in *com.javatunes* and its sub-packages
    - You may also include filters in the component-scan element *
  - No explicit bean definitions shown - but you can also have them
  - XML configuration overrides annotation-based configuration

```
<?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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">

     <context:component-scan base-package="com.javatunes"/>

</beans> <!-- tag configuration example -->
```

Other Stereotype Annotations

- Other stereotype annotations that are detected in auto-scanning include:
  - ***@Controller***: Class is a controller (e.g. Web controller)
  - ***@Repository***: Class is a repository (e.g. DAO)
  - ***@Service***: Class is a stateless service (e.g. service façade)
    - All are specializations of *@Component* indicating some additional functionality
- You can define your own annotations
  - Include *@Component* in their definition to detect them with auto-scanning
- You can declare the bean scope using ***@Scope***
  - e.g. *@Scope("singleton")* - which is the default

@Resource - Additional Uses

- *@Resource* usable to resolve well known dependencies
  - The annotated resource is injected by the container, e.g.

```
@Resource
private ApplicationContext context;
```

- *@Resource* usable without a name element for bean injection
  - Container first tries to resolve using the name of the field
  - Next, tries to resolve using type matching
  - Below, the container first tries to resolve via *name=info*, and if that fails tries to resolve by type with an *InfoSource* bean

```
@Resource
private InfoSource info;
```

- *@Resource* also usable to inject method arguments
  - Optional elements give info on resolving dependency

@Autowired

- ***@Autowired*** provides for annotation-based autowiring
  - Mainly for type-driven injection
- At bottom, we show a simple autowiring example
  - The bean of type *InfoSource* is injected into the *info* field
  - Could annotate the setter method with almost the same result *
- Can be applied to :
  - **Constructors, fields, setter methods**: Normal injection
  - **To a method** with arbitrary name and arguments: All arguments are autowired with a matching bean (see notes)
    - These "config" methods called after constructor/field injection
  - **Collection or Array**: Injects all beans of a type

```
import  org.springframework.beans.factory.annotation.Autowired;
public class JavaInstructor implements Teacher { // Detail omitted
  @Autowired // Injects matching bean (e.g. our springBook bean)
   private InfoSource info;
}  // Remaining detail omitted …
```

@Autowired Examples

- Below, we see autowiring applied to an array
  - Injects ALL beans of type *InfoSource* into the *sources* field
  - *sources* could also be declared a *Collection<InfoSource>*
- At bottom, we see an autowired config method *setSources*
  - The method is called **at bean creation**, and its arguments injected

```
public class JavaInstructor implements Teacher {
   @Autowired // Injects all beans of type InfoSource
   private InfoSource[] sources;
}   // Most detail omitted …
```

```
public class JavaInstructor implements Teacher {
   private InfoSource info;
   private ExamSource tester: // Assume a bean of this type declared

   @Autowired
   public void setSources (InfoSource infoIn, ExamSource examIn) {
      info=InfoIn;
      tester=examIn
   }
}   // Most detail omitted …
```

Qualifiers

- Spring's ***@Qualifier*** gives you some control over injection
  - You associate a qualifier with an autowire, to narrow the type matches
  - Assume we had multiple beans implementing *InfoSource*
  - We could add a qualifier element to one, as shown below
  - In our autowiring, we could then specify the qualifier to narrow the injection, as shown at bottom

```
import org.springframework.beans.factory.annotation.Qualifier;
@Component
@Qualifier("Java")
public class SpringCourseBook implements InfoSource {
   public String getData() {
      return "Dependencies are not so cool";
   }
}
```

```
public class JavaInstructor implements Teacher {
   @Autowired
   @Qualifier("Java") // Narrow injection to bean with Java qualifier
   private InfoSource info;
}  // Most detail omitted, including imports …
```

Lifecycle Annotations

- Methods annotated with lifecycle annotations are called during the bean instance lifecycle (creation/destruction)
  - ***@PostConstruct***: Called after instance is created
  - ***@PreDestroy***: Called before the container destroys instance
    - Standard Java EE annotation in ***javax.annotation*** package
  - Can be used for resource management where needed

```
// import/package statements not shown
@Component("springGuru")
public class JavaInstructor implements Teacher { // Detail omitted
   @Resource(name="springBook")
   private InfoSource info; // Don't even need a setter
   @PostConstruct
   public void init() { /* … */ }
   @PreDestroy
   public void cleanUp { /* … */ }
}
```

Annotation Configuration - Pro / Con

- Pros:
  - Can make configuration **simpler and faster** - configuration closer to bean definition, easier to manage than XML
  - **Annotations**, plus **component scanning**, plus **autowiring** can make configuration much smaller and easier
  - Very good for **quick prototyping**
  - **Refactoring** can be easier
- Cons:
  - **More tightly coupled components**, config changes **need recompile**
  - **Only one bean instance per type** can be configured
  - **Configuration is scattered** among many different files
    - Harder to manage and get an overall view of structure (tool support needed)
    - With XML, configuration is externalized and centralized
  - **Components less reusable** - annotations per type, not per instance
  - Can make **testing harder**

A Note on the XML Configuration

- Spring provides XML Schema based configuration
  - Includes custom namespaces and tags with complex behavior
    - Beyond simple bean definitions and injection
  - e.g. the *context:* namespace we just used
- The Spring namespaces include:
  - ***aop***: Configures AOP support
  - ***beans***: The standard bean tags we've seen already
  - ***context***: Configures ApplicationContext related things
  - ***jee***: JEE related configuration such as looking up a JNDI object
  - ***jms***: Configures JMS related beans
  - ***lang***: Exposing objects written in language like JRuby as beans
  - ***tool***: Adds tooling-specific metadata
  - ***tx***: Configures transaction support
  - ***util***: Common, utility configuration issues

A Brief Note on Annotations

- Standard Java mechanism to **add metadata** to source code
  - They're like comments – except the compiler is aware of them
- Uses @ for declaration and use of annotation types
  - There is a syntax for declaring annotations, using them in declarations, a class file representation, and an API to read them
- Annotations do not directly affect program semantics
  - i.e., they are not executable code
- Annotations affect how tools treat the annotated declarations
  - Which may affect the semantics of a running program
  - For instance, the Spring container reads the annotations on ***@Component*** annotated beans
  - Based on that, it treats it as a bean definition

Annotation Definition Example

- Most programmers won't have to define one - but it's easy
  - Here's a simplified version of the *@Repository* declaration
  - An at-sign (@) precedes the interface keyword
  - Each method declaration defines an *element* of the annotation
  - The simplified *Repository* annotation below has one element
    - *value*
    - If you supply an unnamed element to an annotation, then it is assigned to the "*value*" element (this is part of the Java 5 spec)

```
@Component // Signifies that this is a Component
public @interface Repository {
  //The name to use for the bean in the Spring container
  String value() default "";
}
```

Lab 3.3 – Spring Annotations

Lab 3.3 – Using Annotations

Lab

- **Overview**: In this lab, we will use the Spring annotations to configure all our beans, and any dependencies
  - We'll need to annotate all our beans
  - We'll need to configure the container to recognize the DI annotations, and do the auto-scanning
  - The lab is based on the earlier *TeachMeSpring* labs
- **Objectives**:
  - Write a small application using Spring annotations
- **Builds on previous labs**: none
- **Approximate Time**: 25-45 minutes (depending on optional parts)

Lab Preparation

Lab

- The new lab folder where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab03.3

Tasks to Perform

- Close all open files
- Create a new Java project called Lab03.3 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**

- **Open** the file ***applicationContext.xml*** for editing
  - Note how the context namespace is already included
  - Add in the XML ***component-scan*** elements (example in earlier slides)
  - Specify the scanning of the ***com.javatunes*** package

Add Annotations and Run

Lab

Tasks to Perform

- **Open** the file *SpringCourseBook.java* for editing
  - Add the ***@Component*** annotation, and specify a bean name of ***springBook*** (you'll need to import *Component*)
- **Open** the file *JavaInstructor.java* for editing
  - Add the ***@Component*** annotation, and specify a bean name of ***springGuru*** (you'll need to import *Component*)
  - Add the ***@Resource*** annotation to the info field, with a resource name of ***springBook*** (you'll need to import *Resource*)
- **Review and run** the program in ***TeachMeSpring.java ****
  - It should run as previously - all the DI and bean lookup works as before

Autowiring

Lab

Tasks to Perform

- In *JavaInstructor*, remove *@Resource*, and annotate the info field so it is autowired (this should work fine)
  - Run the program again to see that it works
- **Optional**: See how autowiring can fail
  - We supply another *InfoSource – RubyCourseBook*
  - Open this class and annotate it as a component
  - Run the program again – it should fail with an exception because you now have multiple candidate *InfoSource* beans for an autowiring
- **Optional**: Use qualifiers
  - Add a qualifier to *SpringCourseBook*, and to the info autowiriring
  - Run the program again – it should work how *
- **Optional**: Try adding in some of the lifecycle annotations
  - They can just print out some text to show when they're called
  - Run the program and look at the output

STOP

Java-based Configuration

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

Java Configuration Overview

- Spring 3 introduced a Java-based Spring configuration,
  - It decouples the configuration from the bean definitions
  - *@Configuration* annotated classes contain configuration info
    - They replace XML definition files, and similarly **decouple the configuration** from the bean class definitions
  - *@Bean* annotated methods that return a bean define the bean
  - This example configures the same bean as the XML at bottom

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class TeachConfig {

  @Bean
  public Teacher springGuru() {
    return new JavaInstructor();
  }
}
```

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

Using Java-based Configuration

- *AnnotationConfigApplicationContext* is used to recognize *@Configuration* classes
  - At its most basic, you simply pass in the configuration class
    - Can pass in multiple classes also (comma separated)
  - Note how the bean lookup is exactly the same
  - This context also recognizes *@Component* stereotypes

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TeachMeSpring {

  public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx =
      new AnnotationConfigApplicationContext(TeachConfig.class);
      Teacher teacher = ctx.getBean("springGuru", Teacher.class);
      teacher.teach();
  }
}
```

Dependency Injection

- Wiring a bean is done by referring to its *@Bean* method
  - The example below wires the *springBook* bean into the *springGuru*'s info property – the same as the XML at bottom

```
@Configuration
public class TeachConfig {

   @Bean
   public Teacher springGuru() {
      JavaInstructor ji = new JavaInstructor();
      ji.setInfo(springBook());
      return ji;
   }

   @Bean
   public InfoSource springBook() {
      return new SpringCourseBook();
   }
}  // This is equivalent to the XML config below
```

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

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

More on How it Works

- An *@Configuration* class **is metadata**
  - Much like an XML file – just a different format
- *@Bean* methods **aren't called like normal methods**
  - Interpreted as metadata indicating a bean definition
  - "Calling" the method in a config class specifies an injection
- Wiring in another *springBook* bean (see below) **uses the same bean instance** (since by default a bean is a singleton)
  - Confusing if looked at as regular Java
    - It seems to create a new instance – but it doesn't (see notes)
  - Remember – this is **metadata**, not executable code

```
// Rest of @Configuration class not shown (same as on previous slide)
@Bean
public Teacher javaWhiz() {
  JavaInstructor ji = new JavaInstructor();
  ji.setInfo(springBook());
  return ji;
}
```

Dependencies Between Configuration Classes

- *@Configuration* classes can have **dependencies** on beans from other *@Configuration* classes
  - You can inject them, like any defined bean, as shown below
    - *TeachConfig* uses the *springBook* bean

```
@Configuration
public class TeachConfig {
   @Autowired // or @Resource(name="springBook")
   private InfoSource springBook;
   @Bean
   public Teacher springGuru() {
      JavaInstructor ji = new JavaInstructor();
      ji.setInfo(springBook);
      return ji;
   }
}
```

```
@Configuration
public class BookConfig {
   @Bean
   public InfoSource springBook() { return new SpringCourseBook(); }
}
```

Other Usage Scenarios

- An *@Configuration* class can import another
  - For example, to import a class *BookConfig*:

```
@Configuration
@Import(BookConfig.class)
public class TeachConfig { ... }
```

- An *@Configuration* class can import an XML config file
  - For example, to import a file *otherConfig.xml*:

```
@Configuration
@ImportResource("classpath:otherConfig.xml")
public class TeachConfig { ... }
```

Classpath Scanning

- *AnnotationConfigApplicationContext* can use classpath scanning, as shown below
  - *scan()* takes a package name (or multiple names), just like *context:component-scan*
  - Note: You must call *refresh()* on the context after scanning, so that the beans are initialized
- An *@Configuration* class can be declared in XML (see notes)
  - So you can use an *@Configuration* centric or XML centric approach, and mix in the other approach

```
public class TeachMeSpring {

  public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx =
                                  new     AnnotationConfigApplicationContext();
    ctx.scan("com.javatunes.teach");
    ctx.refresh();
    // Rest of code the same ...
  }
}
```

Other @Bean Capabiliites

- A bean can specify a scope other than the default singleton
  - Using *@Scope*, as shown below

```
@Bean
@Scope("prototype")
public Teacher springGuru() { /* ... */ }
```

- A bean can specify a name

```
@Bean(name="springGuru") // Bean is named springGuru
public Teacher getTeacher() { /* ... */ }
```

- You can even have multiple aliases using a string array

```
@Bean(name={ "springGuru", "javaWhiz" })
```

  - The example uses the standard Java array initialization syntax

- You can specify init / destroy methods

```
@Bean(initMethod="init", destroyMethod="cleanup")
```

  - Called at appropriate times when instance created

Java-based Configuration - Pro / Con

- Pros:
  - **No XML**, Java used for configuration, fairly compact and readable
  - Configuration is **external to bean code**
  - Full control over object creation (e.g. two beans of same type is easy)
  - Refactoring friendly – uses normal Java tools
  - Type safety – the compiler checks most things (unlike in XML, where there are a lot of strings and many errors, like misspelling a classname, are often not caught until runtime)
- Cons:
  - Yet another way of configuring Spring that developers have to learn
  - Somewhat obscure syntax
  - Changes require **recompilation**
  - Relatively new, and may still need some time for tooling and for refining its capabilities

[Optional] Lab 3.4 – Java-based Config

Lab 3.4 – Java-based Configuration

Lab

- **Overview**: In this lab, we will use Spring's new Java-based configuration to configure all our beans, and dependencies
  - We'll create a separate *@Configuration* class to do the configuration
  - The lab is based on the earlier *TeachMeSpring* labs
- **Objectives**:
  - Write a small application using Java-based configuration
- **Builds on previous labs**: none
- **Approximate Time**: 20-45 minutes (depending on optional parts)

Lab Preparation

- The new root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab03.4

Tasks to Perform

- **Close** all open files and projects

- Create a new Java project called Lab03.4 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**

- **Open** the file ***applicationContext.xml*** for review
  - Note the ***annotation-config*** elements
  - Nothing you need to do in this file

Add Annotations and Run

Lab

Tasks to Perform

- **Open** the file *TeachConfig.java* for editing
  - Add the ***@Configuration*** annotation to it, and add code to specify:
    - An *InfoSource* bean named ***springBook*** using *SpringCourseBook* to create the bean
    - A *Teacher* bean named ***springGuru*** using *JavaInstructor* to create the bean – you will need to set its *info* property using the *springBook* bean
- **Open** the file *TeachMeSpring.java* for editing
  - Create an *AnnotationConfigApplicationContext* that uses TeachConfig for its configuration
- **Run** the program in ***TeachMeSpring.java***
  - It should run as previously - all the DI and bean lookup works as before

Optional Things to Try

Lab

[Optional] Tasks to Perform

- In *TeachConfig*, add an init method declaration for the *JavaInstructor* bean def
  - Specify the init() that already exists in *JavaInstructor*
  - Run the program and see that it is called (it prints a console message)
- In TeachMeSpring.java, use classpath scanning via the *AnnotationConfigApplicationContext*
  - Rather than explicitly naming the config file
  - Scan the *com.javatunes* package – remember to call *refresh()*
  - Run the program to see it work
- Create a new config class named *BookConfig*
  - Move the declaration of the *springBook* bean into this class
  - Inject the *springBook* bean into the *TeachConfig* class for use in creating the *JavaInstructor*
  - Run your program and see that it works

STOP

Maven and Spring

ApplicationContext

Bean Scope and Lifecycle

Annotation Driven Configuration

Java-based Configuration

Maven and Spring

About Maven

- **Maven**: Tool for building and managing Java projects
  - **POM** (Project Object Model - an XML file) used for configuration
  - Maven automates the build process (and much more)
- Maven can **automatically download dependencies**
  - As configured in the POM, and downloaded as part of build
  - Default: Download via http from the **maven central repository**

    http://repo.maven.apache.org/maven2/
- Maven is the **de-facto standard for Spring** builds
  - Well supported by Spring
  - **No other easy way** to manage Spring dependencies (see notes)
- We **supplied the dependencies** for most of the labs
  - So you don't have to deal with maven in them
  - We used maven to figure out the needed dependencies

How We'll Work With Maven

- **Goal**: Understand how maven supplies Spring dependencies
  - This is important for Spring developers to be familiar with
  - We'll work within the context of our lab environment
  - We will NOT give an in-depth coverage of maven
- We'll use maven for **dependency management only**
  - We'll continue to use Eclipse for building projects and running programs
  - Maven has related capabilities (and more), which we won't use
- **We'll use the Maven for Eclipse** plugin to work with maven
  - We won't use maven directly (e.g. from the command line)
- We won't require maven for most of the labs
  - As it may not be available in all lab environments

The POM (Project Object Model)

- The **POM** is an XML file containing information and configuration details about a maven project
  - Typically in a file ***pom.xml***
- The example below contains some required POM properties
  - It also includes a single dependency on a Spring module
  - We'll cover the details next

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatunes.spring</groupId>
  <artifactId>Lab03.5</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.2.2.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

POM - Required Elements

- The POM elements in the example below are required
  - ***project***: The root element
    - This element includes namespaces - omitted in the example
  - ***modelVersion***: The model version maven should use
    - Should be 4.0.0 for current versions of maven
  - ***groupId***: The id of your project's group
  - ***artifactId***: The id of the project
  - ***version***: The version of the project
  - ***groupId:artifactId:version*** uniquely identifies the project
    - This is called the fully qualified artifact name

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javatunes.spring</groupId>
  <artifactId>Lab03.5</artifactId>
  <version>1.0</version>
</project>
```

POM - External Dependencies

- The POM below declares a single external dependency
  - On ***spring-context*** (One of the core Spring artifacts)
    - Referenced via its fully qualified artifact name
    - Note: We use a property for the Spring version to change it easily
  - Based on this POM, maven will **automatically include** the Spring context jar (and its dependencies) in your project

```
<project> <!-- Previous detail omitted -->

  <properties>
    <org.springframework.version>3.2.2.RELEASE</org.springframework.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${org.springframework.version}</version>
    </dependency>
  </dependencies>

</project>
```

Maven Artifacts for Spring

- Listed below are some common Spring artifacts
  - They are in the maven repository with GroupId *org.springframework*
    - Can be used as dependencies in your POMs

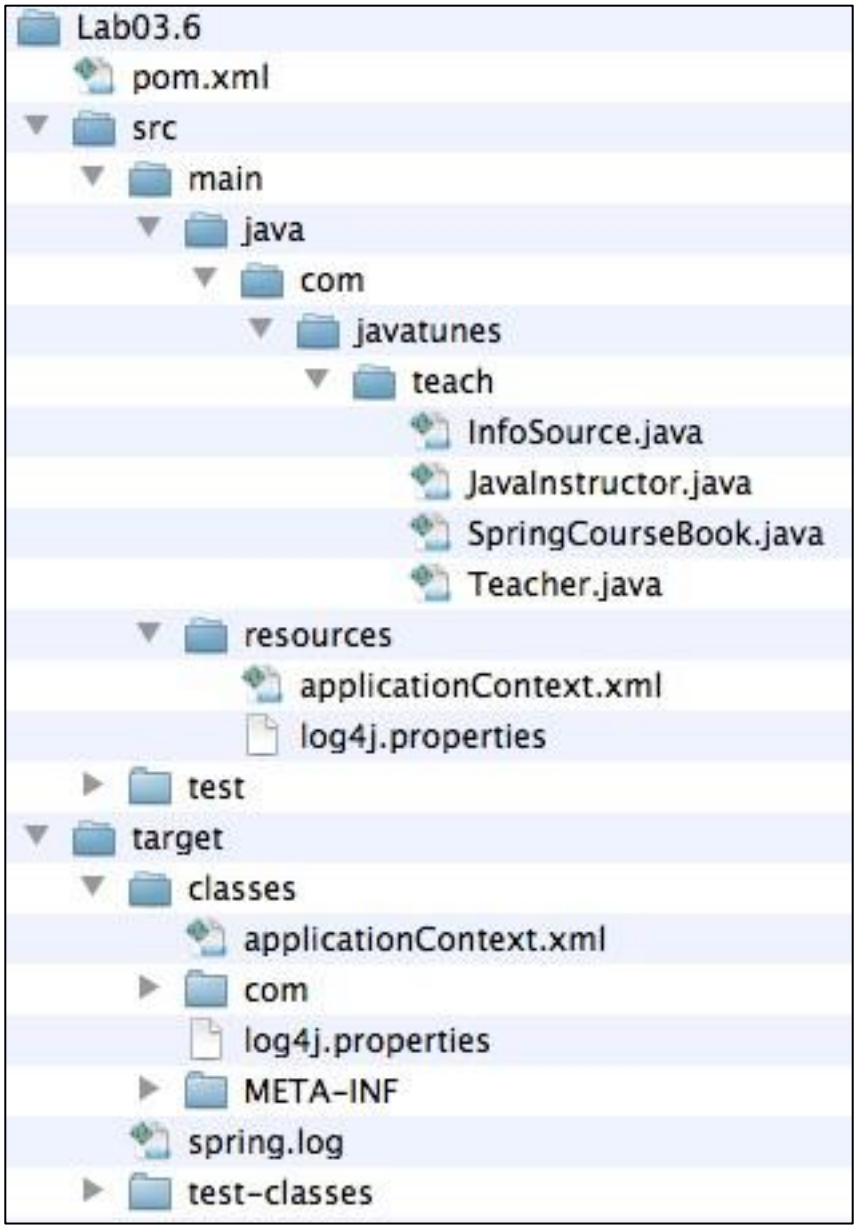
- **spring-aop**: Aspect Oriented Programming (AOP) Framework
- **spring-beans**: Bean Factory and JavaBeans utilities
- **spring-context**: Application Context
- **spring-core**: Core utilities used by other modules
- **spring-expression**: Expression Language (depends on spring-core)
- **spring-instrument**: Instrumentation for AOP
- **spring-jdbc**: JDBC Data Access Library (e.g. JdbcTemplate)
- **spring-orm**: Object-to-Relation-Mapping (ORM) integration e.g. JPA
- **spring-tx**: Spring transaction management
- **spring-web**: Web application development utilities
- **spring-webmvc**: Spring MVC for Servlet Environments

Repositories

- Maven looks for external dependencies in a repository
  - A repository holds build artifacts and dependencies
- By default, maven looks in your local repository
  - In ***~/.m2/repository*** - the default location
- Dependencies not in the local repository are downloaded from a remote repository
  - They are then put in your local repository
  - Default remote repository, "The Maven Central Repository" at: ***http://repo.maven.apache.org/maven2/***
- You can configure all of this differently
  - The details are beyond the scope of this course

Maven Project Structure

- The standard structure for a standalone Java project in Maven
  - **Root folder**: *pom.xml*
  - ***src/main/java***: Java source files
  - ***src/main/resources***: Resources that should be on classpath (e.g. config files like *applicationContext.xml*)
  - ***src/test***: Test source files
    - We put our main class to run our Spring programs under this folder
  - ***target***: Build output
- Web projects have an additional folder - for JSPs, html, etc
  - ***src/main/webapp***



Eclipse / Maven Integration

- Done with **Maven Integration for Eclipse** plugins
  - Consists of two plugins - m2e and m2e-wtp
  - Available via the Eclipse update sites:

  http://download.eclipse.org/technology/m2e/releases

  http://download.eclipse.org/m2e-wtp/releases/

- Capabilities include:
  - **Dependency management** for Eclipse build path based on Maven's *pom.xml*
  - **Automatic downloading** of the required dependencies from the remote Maven repositories
  - Quick fixes in the Java editor for looking up required dependencies/jars by the class or package name
  - We'll use a few of these capabilities in the labs

The Lab

- The lab allows you to see maven at work
  - It will be used to download all your project dependencies and include them on the build path
  - There is no coding in the lab - it's solely to work with maven
- You'll need to do the following
  - Install the m2e-wtp plugin (which also installs the m2e plugin)
  - Create a Java project
  - Convert the project to a maven-based project
  - Set up *pom.xml* (which we provide) and update the project
- **You need internet access** to the Eclipse update sites, and the maven central repository
  - Otherwise, you'll need to skip the lab

[Optional] Lab 3.5 – Using Maven

Lab 3.5 – Using Maven

Lab

- **Overview**: In this lab, we will create a Java Project that uses maven for its dependency management
  - As part of that, we'll install the Eclipse m2e and m2e-wtp plugins
  - We'll also create a project, and convert it to a maven project

- **Objectives**:
  - Become familiar with using Maven from within Eclipse
  - Use maven for Spring dependency management

- **Builds on previous labs**: None

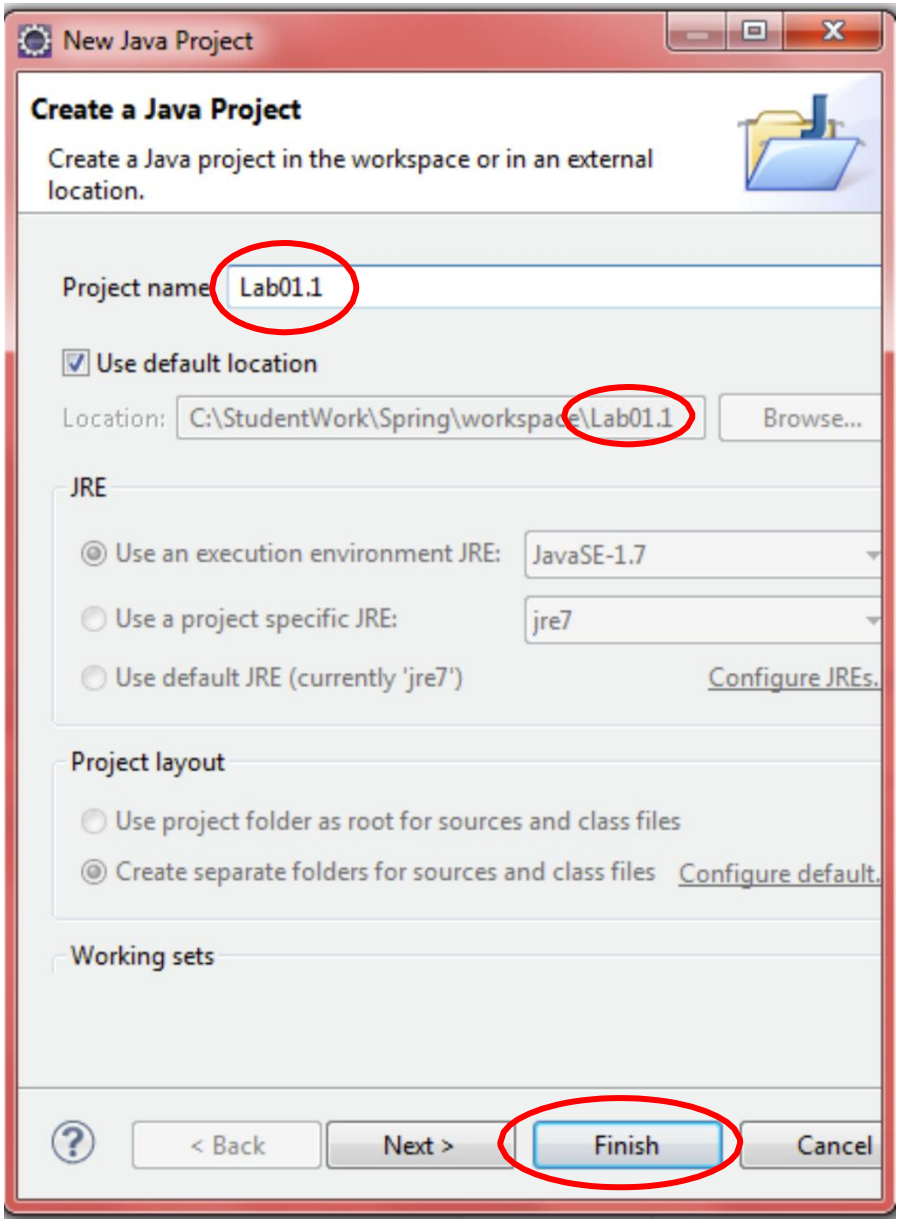
- **Approximate Time**: 35-45 minutes

Create a Project for our Application

Lab

Tasks to Perform

- Create a **Java Project** *
  - Call the project **Lab03.5**
  - Eclipse will then automatically set the project folder to *Lab03.5*
    - This folder already exists in the workspace
    - It contains a Java project laid out for Eclipse / Maven
  - Click **Finish**
- Ignore any errors
  - We still need to configure the Spring dependencies (via maven)



Add Maven Support to Eclipse

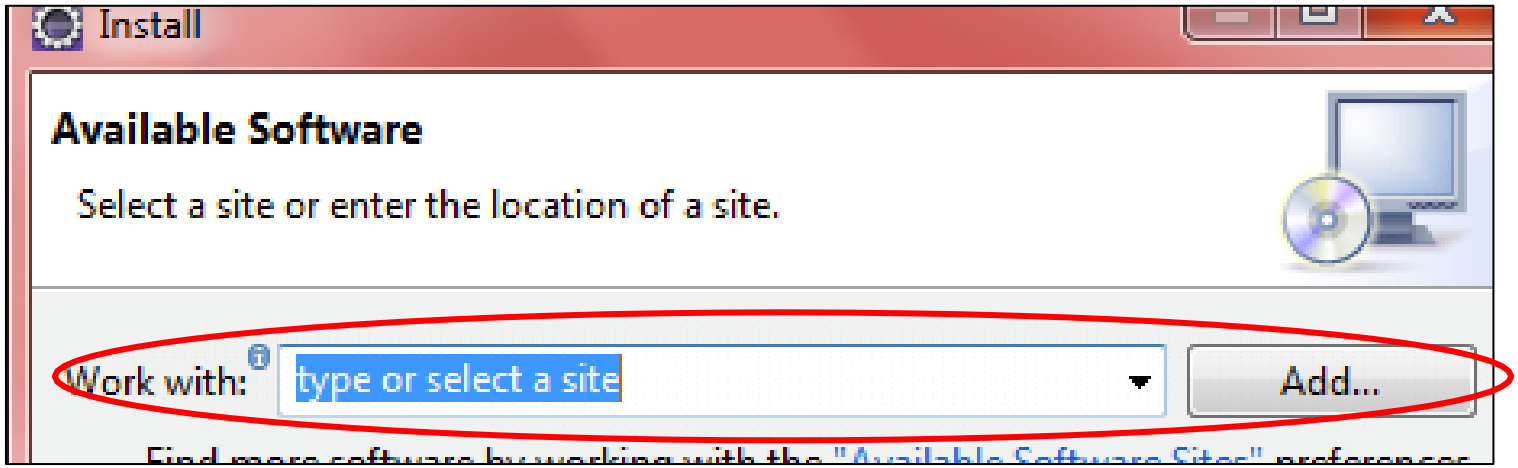
Lab

- Done by installing the m2e and m2e-wtp plugins
- If **unable to install m2e** or **access the maven central repository**, you'll need to skip the lab

Tasks to Perform

- See notes if **using a proxy server**
- Go to **Help | Install New Software …**
  - In the resulting page, click the **Add…** button to add the maven site
  - Call the site Maven, and set the URL to the below (see notes):

    http://download.eclipse.org/m2e-wtp/releases/



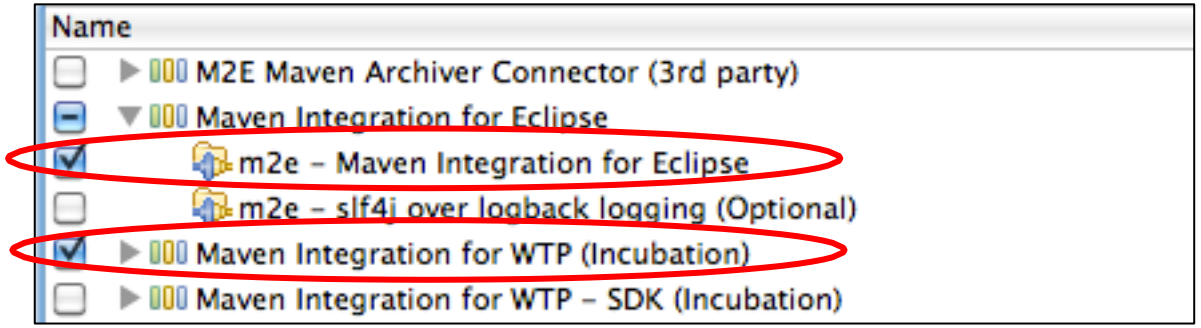
Add Maven Support to Eclipse

Lab

Tasks to Perform

- Click **OK**, expand the Maven Integration for Eclipse plugin, and check the **m2e** item (see below)
- Check the **Maven Integration for WTP** checkbox also
- Click **Next** until you see a license agreement - accept it, click **Finish**
- The m2e and m2e-wtp plugins for Eclipse should now install
- **Restart** Eclipse when prompted

- Go to **Window | Preferences | Maven**
  - Uncheck **Download repository index updates on startup**
  - Which can take a lot of time, and is not needed for the class



Add Maven Support to Project

Lab

Tasks to Perform

- The supplied files are already structured as a Maven Java project
  - We must add in Eclipse support for maven, as follows
- Right click on the project in Project Explorer, select **Configure | Convert to Maven Project**, click **Finish**
  - You can leave the default values for Group Id, etc. as we will replace the generated *pom.xml* with our own
- In the project, **delete the generated *pom.xml*** (in the project root)
  - **Rename** the supplied *pom.xml.txt* file to *pom.xml* (Right click, Rename)
- Open *pom.xml* for review (see next slide for excerpts)
  - Note that all **projects are configured to use Java 7** in the *maven-compiler-plugin* configuration section
  - Note the dependency elements for spring and log4j
  - Right click on project, select **Maven | Update Project** (pick up changes)

Review pom.xml

Lab

Tasks to Perform

- Open *pom.xml* for review
  - Select the *pom.xml* tab (excerpts shown at right)
  - Note the dependency on **spring-context**
  - Pulls in the spring context jar and all its dependencies
  - There is also a dependency on log4j (not shown here)
- Note the maven-compiler-plugin config (see notes)
  - **If not using Java 7**, then **revise** the source/target values to your Java level (e.g. 1.6 for Java 6)

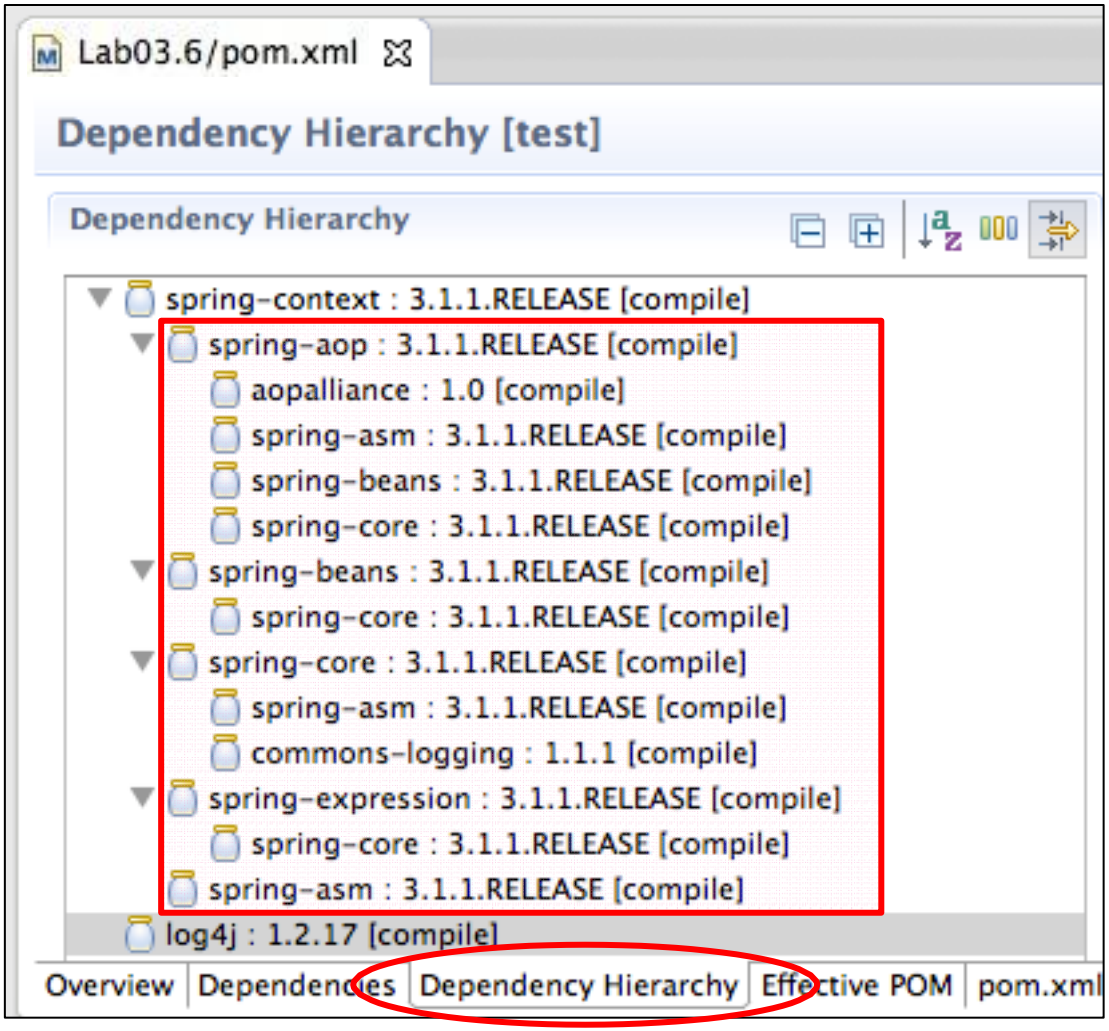
```
<properties>
  <org.springframework.version>
        3.2.2.RELEASE
   </org.springframework.version>
</properties>

<dependencies>
  <dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>
      ${org.springframework.version}
    </version>
  </dependency> <!-- detail omitted -->
<dependencies>

<plugin>
  <artifactId>
      maven-compiler-plugin
   </artifactId>
  <version>2.3.2</version>
  <configuration>
   <source>1.7</source>
   <target>1.7</target>
  </configuration>
</plugin>
```

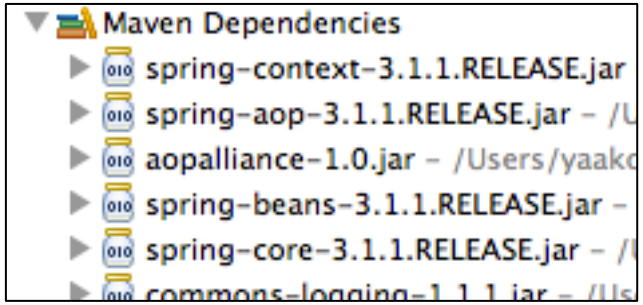
Maven Only: Review pom.xml

Lab

Tasks to Perform

- Click the **Dependency Hierarchy** tab of *pom.xml*
  - Note the jars that are dependencies of spring-context (at right)
- You can also look at the **Maven Dependencies** node in Package Explorer
  - You'll see the jars in the project that were pulled in by maven (below)

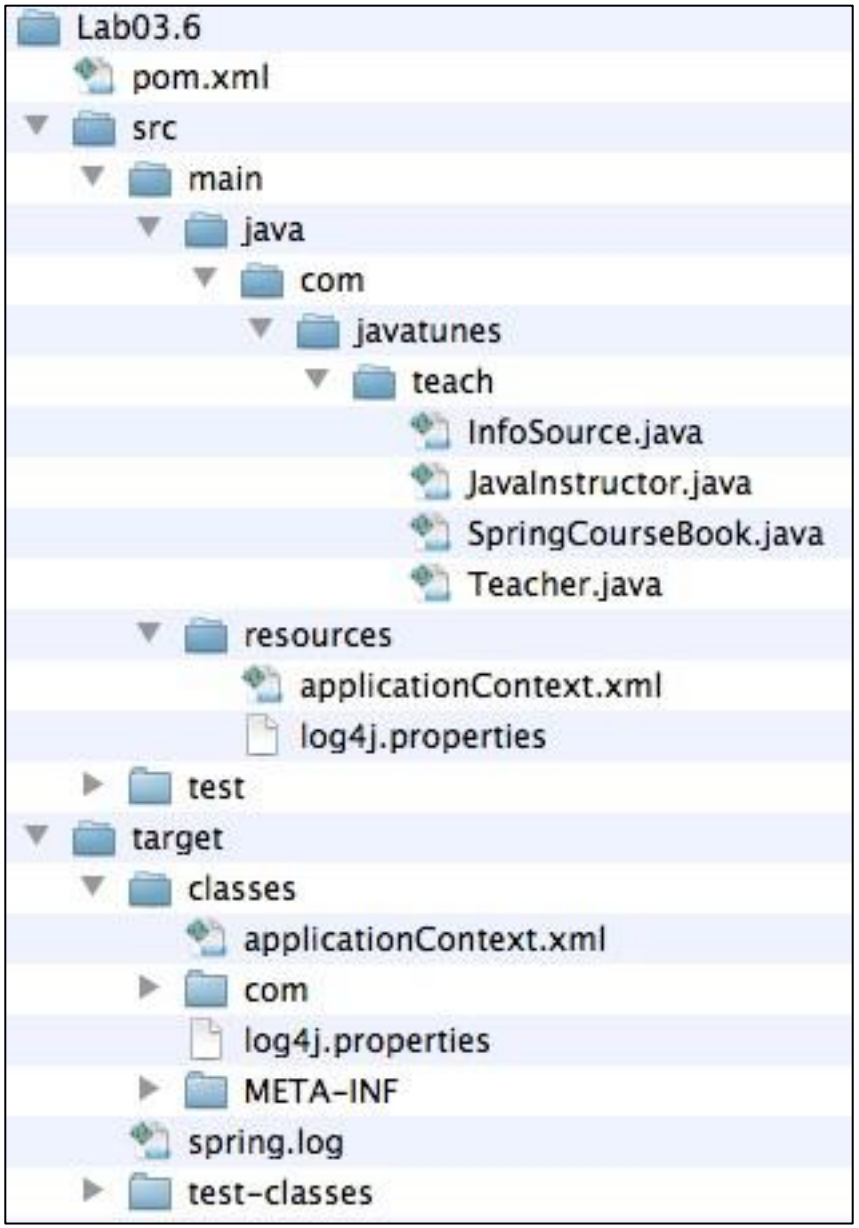




Project Structure (Maven/Eclipse)

Lab

- Follows maven/Eclipse Java project structure
  - **Root folder**: *pom.xml*
  - ***src/main/java***: Java source files
  - ***src/main/resources***: Resources that should be on classpath (e.g. config files like *applicationContext.xml*)
  - ***src/test***: Test source files
    - We put our main class to run our Spring programs under this folder
  - ***target***: Build output (only if using maven)

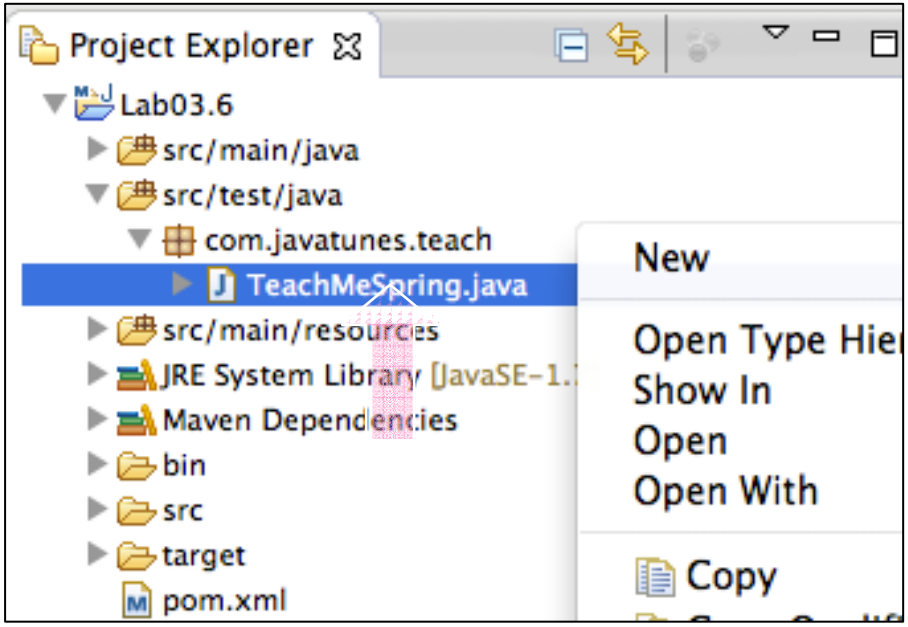


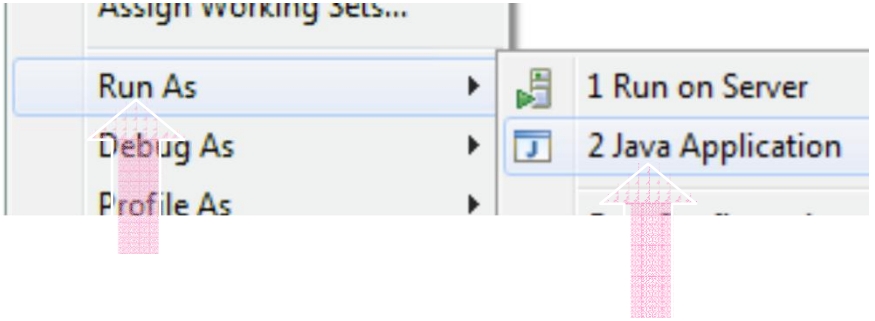
Running the Lab

Lab

Tasks to Perform

- We use Eclipse to run a program we supply in *TeachMeSpring.java*
  - This is basically unchanged from non-maven projects
- After a clean build do the following
  - In the **Package Explorer** pane, right click on *TeachMeSpring.java* (in package com.javatunes.teach under the src/test/java folder)
  - Select:
    Run As -> Java Application
  - This will automatically find the main method in *TechMeSpring* and run it for you





Program Output

Lab

- You should see output in the Eclipse Console view showing the results of running the program
  - This is the same as with non-maven projects
- That's it - you've used maven to manage your dependencies
  - Note that you didn't need to add in the Spring user library

Problems @ Javadoc Declaration Console

```
<terminated> TeachMeSpring [Java Application] /Library/Java/JavaVirtualMachines/1.7.0u4.jdk/Contents/Home/bin/java (Jan 16, 2013 2:0
2013-01-16 14:04:53,489 INFO [org.springframework.context.support.ClassPathXmlApplicationContext] - Refresh
2013-01-16 14:04:53,551 INFO [org.springframework.beans.factory.xml.XmlBeanDefinitionReader] - Loading XML
2013-01-16 14:04:53,819 INFO [org.springframework.beans.factory.support.DefaultListableBeanFactory] - Pre-i
Dependencies are not so cool
```

STOP

Review Questions

- What is an *ApplicationContext* and how is it different from a *BeanFactory*
- What capabilities does Spring give you to access your program resources?
- What is the scope of a bean?
  - What are the different scopes that Spring gives you for working with beans (not including web-based scopes)?
  - What is an inner bean?
- Describe the lifecycle of a bean managed by the Spring container - What are lifecycle events?
- How does Spring use annotations for defining beans and DI?
  - What other annotations does it support?

Lesson Summary

- The Spring *ApplicationContext* extends *BeanFactory*, and provides the full Spring container functionality
  - Includes the ability to help you access program resources, work with resource bundles, load multiple contexts, and publish events to beans
  - *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* are two common implementations of this interface used in standalone applications

- Spring also provides a complete set of resource access classes
  - Provides resource access (files, urls, etc) in a flexible and powerful way from different locations such as the file system, the classpath, or via a URL
  - Common implementations are *ClassPathResource*, *FileSystemResource*, and *UrlResource*

Lesson Summary

- Spring easily defines both **singleton** and **prototype** scope beans
  - **Singleton bean**: Container creates a single bean instance for a given id
  - **Prototype bean**: Container creates a new instance for each request of a bean with a given id
  - **Inner** beans are associated with the containing bean
    - They are anonymous, and always have prototype scope

- Beans have a sophisticated lifecycle within the Spring container
  - Many lifecycle steps, e.g. instantiating the bean, populating the properties, and calling Spring specific callbacks
  - Spring provides interfaces, such as *ApplicationContextAware*, that allow beans to participate in their lifecycle
  - The interfaces define callback methods from the container that are called at the appropriate time in the lifecycle

Lesson Summary

- Spring provides ***@Component*** to declare beans via annotations
  - It also supports ***@Resource*** for Dependency Injection
  - Supports ***@PostConstruct*** / ***@PreDestroy*** lifecycle callbacks
- **Java-based configuration** was added in Spring 3
  - Can eliminate almost all XML configuration
  - Uses ***@Configuration*** on configuration classes
  - Uses ***@Bean*** on methods that are bean definitions
  - It is a flexible and powerful alternative to XML configuration
- **Maven**, a tool for building and managing Java projects, can be used to manage spring dependencies
  - The **POM** (*pom.xml*) is used for configuration of your project
  - Maven can then download all dependencies and build a project
  - Maven can be integrated with Eclipse via Eclipse plugins

Module 4: Database Access With Spring

Overview

Using Spring with Hibernate

Using Spring with JPA

Note: The Hibernate and JPA sections are independent. You can skip whichever you're not using

Lesson Objectives

- Understand the DAO pattern, and how Spring supports it
- Be familiar with Spring's datasource support
- Be familiar with the Spring Support for Hibernate and JPA

Overview

Overview

Using Spring with Hibernate

Using Spring with JPA

DAO Support

- A **DAO** (**Data Access Object**) can encapsulate DB code
  - One typical approach to deal with the complexity
  - Puts DB functionality **in one place**, making it easier to manage
  - **Encapsulates** DB access
    - Internals not accessible to clients
  - **Insulates your other code** from database details
    - Making clients much simpler
- Spring has extensive support for DAOs
  - We'll focus on Hibernate and JPA-based DAOs
  - We won't cover plain JDBC-based DAOs - no longer widely used
- We'll also look at Spring's datasource support

Datasources

- *javax.sql.DataSource* is standard Java way to manage DB connections
  - It's a connection factory for the underlying physical DB
  - It's configured with connection information
  - Provides connections to client code (via *getConnection()*)

- Spring classes for working with *DataSource*s include:
  - *DriverManagerDataSource*: Simple non-connection pooled implementation of *DataSource*
    - Suitable for testing/development
    - Includes properties for common configuration – e.g. *driverClassName*, *url*, *username*, *password*
  - *JndiObjectFactoryBean*: For general purpose JNDI look up
    - Can be used to look up an existing *DataSource* in JNDI

Configuring the DataSource

- Below, we configure a *DataSource*
  - The ***personnelDataSource*** bean contains DB connection info *
    - It is a Spring ***DriverManagerDataSource***
  - The ***employeeDAO*** bean has the datasource injected
    - It is a ***JdbcEmployeeDAO***, as shown previously
  - The datasource is typically injected into a DAO

```
<bean id="personnelDataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName"
             value="org.apache.derby.jdbc.ClientDriver"/>
   <property name="url"
             value="jdbc:derby://localhost:1527/PERSONNEL"/>
   <property name="username" value="guest"/>
   <property name="password" value="password"/>
</bean>
```

Looking up a DataSource in JNDI

- Often useful, especially in JEE environments
  - Simple to do with the ***jee*** namespace as shown below
  - The ***resource-ref="true"*** attribute signals Spring to look in the local environment
    - i.e. prefix *java:comp/env* to the JNDI name
  - You can do the same lookup with *JndiObjectFactoryBean* *

```
<?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:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-3.1.xsd">

    <jee:jndi-lookup id="personnelDataSource" jndi-name="/jdbc/personnelDS"
            resource-ref="true">

</beans>
```

[Optional] Using Spring with Hibernate

Overview

Using Spring with Hibernate

Using Spring with JPA

Hibernate Overview

- Hibernate is an Open Source, ORM (Object-Relational Mapping) framework for Java
  - It provides **ORM capability** for using a **Java OO domain model** to manage a relational database
  - Hibernate provides a metadata driven approach to ORM

- Global configuration done in a Hibernate configuration file
  - Typically called ***hibernate.cfg.xml***
  - For example, you usually configure a ***Module Factory*** representing a single database in this configuration fil
- Mappings for individual classes appear in their own mapping file, for instance ***Employee.hbm.xml***
  - In this mapping file you would configure the mapping of the *Employee* class to the database

Typical Hibernate Configuration File

- Here we show a simplified configuration file
  - To illustrate what it looks like
    - We assume that you already know basic Hibernate

```
<!-- Other detail omitted -->
< Module
-factory>
  <!-- Database Connection Settings -->
  <property  name="hibernate.connection.username">guest</property>
  <property  name="hibernate.connection.password">password</property>
  <property  name="hibernate.connection.url">
      jdbc:derby://localhost:1527/PERSONNEL</property>
  <property  name="hibernate.onnection.driver_class">
      org.apache.derby.jdbc.ClientDriver</property>

 <!-- SQL Dialect -->
  <property  name="hibernate.dialect">
      org.hibernate.dialect.DerbyDialect</property>
  <!-- Specify a mapping resource -->
  <mapping    resource="com/javatunes/persist/Employee.hbm.xml"/>/
  Module
  -factory>
```

Using Hibernate Directly

- *Module*
  Factory and *Module*
  are two core Hibernate interfaces
  - *Module*
    Factory is configured to connect to a DB, and gives you
    Module
    s for that DB
  - A **Module**
    is used to interact with the DB
  - We show a (very simplified) example below

- Note that in later examples using Spring we'll ignore transactions
  - We'll get to transactions later

Using Hibernate Directly

```
// much code omitted ...
// Create the Module
factory based on configuration shown previously Module
Factory sf = new Configuration().configure().build Module
Factory();
 Module
s = sf.open Module
();                                    // Create a
Module
s.beginTransaction();
Long id = new Long(1);
Employee e = (Employee)s.get(Event.class,id);
System.out.println("Retreived Employee: " + e.getName());
s.getTransaction().commit();
s.close();
sf.close();
```

Spring Support for Hibernate

- Spring supports Hibernate by providing Hibernate specific versions of the core Spring JDBC classes
  - They provide abstractions of the Hibernate types, and simplify the details of working with them
  - The structure is the same as that used for Spring JDBC
  - Spring 3.1+ supports Hibernate 3 and 4

- The Spring Hibernate 3 packages are:
  - ***org.springframework.orm.hibernate3***: Main package defining integration between Spring and Hibernate
  - ***org.springframework.orm.hibernate3.annotations***: Provides support for Hibernate 3 (JPA) annotations
  - ***org.springframework.orm.hibernate3.support***: Defines support classes for working with Hibernate 3

Local

- ***Local Module***
  FactoryBean: *FactoryBean* that creates a Hibernate *SesssionFactory*
  - This is the usual way to configure a shared Hibernate *Module*
    Factory in a Sping context
  - Useful, even when not using *HibernateTemplate*

- Has properties for Hibernate-based configuration, including:
  - ***dataSource***: DataSource used by the Module Factory
  - ***mappingLocations***: Collection of Hibernate mapping resources, such as *com/javatunes/persist/Employee.hbm.xml*
  - ***hibernateProperties***: Hibernate specific properties, such as *hibernate.dialect*
  - ***configLocation/configLocations***: Location of Hibernate

Local

XML config file(s) for using Hibernate style configuration

Configuring a Hibernate

```
<!-- Datasource config not shown - similar to previous config -->
<bean  id="personnelDataSource">  ...  </bean>

<bean  id="personnel Module
   Factory"  class="org.springframework.orm.hibernate3.Local Module
   FactoryBean">
   <property  name="dataSource"  ref="personnelDataSource"/>
   <property  name="mappingResources">
      <list>
         <value>com/javatunes/persist/Employee.hbm.xml</value>
      </list>
   </property>
   <property name="hibernateProperties">
      <props>
         <prop key="hibernate.dialect">
                org.hibernate.dialect.DerbyDialect</prop>
      </props>
   </property>
</bean>
```

Contextual

- Hibernate 3 introduced contextual Module s
  - *Module Factory.getCurrent Module ()* provides a Module associated with the "current" context
  - Initially this required JTA
    - The Module context is a JTA transaction
  - Hibernate 3.1 added support for Java SE
    - It scopes the Module to a TX using the thread of execution

- Contextual Module s **ease management and propagation of a Hibernate Module**

Contextual

- This was one of *HibernateTemplate*'s main responsibilities
- You can write a Hibernate DAO **without using Spring APIs**
- You don't need a row mapper, since Hibernate does that

Spring Free DAO

- The example below doesn't use the Spring API
  - You can still use Spring to inject the Module factory
  - It does throw Hibernate specific exceptions, not Spring ones
    - *HibernateTemplate* would throw Spring exceptions
- You can use any other Spring capabilities
  - For example, Spring managed transactions (covered later)

```
package com.javatunes.persist;
import org.hibernate. Module                                                    Factory;

public class HibernateEmployeeDAO implements EmployeeDAO {
    private Module                                                    Factory Module
  Factory; // get/set methods not shown

  public Employee searchById(Long id) {
    return (Employee)get Module
        Factory().getCurrent Module
        ().get( Employee.class, id);
    }
}
```

What Approach to Use

- There are multiple approaches to using Spring with Hibernate
  - They are both powerful frameworks, with some overlapping capabilities
- Choose the approach that makes most sense for you
  - For example, if you may switch to a different persistence framework, you can use *HibernateTemplate* to insulate you from the change
  - If you're committed to Hibernate, using contextual Module s directly in the DAO is easy
    - It also does not add any coupling to Spring in the DAO
- You can use Spring DI and transaction support with either approach

[Optional] Using Spring with JPA

Overview

Using Spring with Hibernate

Using Spring with JPA

Template Support for JPA

- JPA: Standard Java ORM mapping specification
  - Similar to Hibernate – uses annotations for mapping
  - An *EntityManager* executes persistent operations
- Spring supports **JPA** similarly to how it supports Hibernate
  - Providing classes that help create JPA based DAOs
  - These are in the package *org.springframework.orm.jpa*
- Spring provides classes for managing/injecting JPA persistence units and entity managers, and we cover these
  - Considered best practice in integrating Spring and JPA
- Old style templates also supported, including *JpaTemplate* and *JpaDaoSupport*
  - These are no longer recommended for Spring / JPA integration

Managing the EntityManager

- Spring supports management and injection of an *EntityManager* (*EM*) and/or *EntityManagerFactory* (*EMF*)

- Multiple ways to accomplish this – via JNDI lookup, or via the two classes below
  - Both in the package *org.springframework.orm.jpa*
  - ***LocalContainerEntityManagerFactoryBean***: Creates a container managed *EMF*
    - Flexible configuration, container managed EM
  - **JNDI lookup**: Looking up of an *EMF* in a Java EE environment using standard Spring/JNDI capabilities
  - ***LocalEntityManagerFactoryBean***: Creates an *EMF* suitable for standalone programs (only)
    - Least flexible in terms of configuration

LocalContainerEntityManagerFactoryBean

- Gives full control over EMF configuration
  - Appropriate for fine-grained customization
  - Supports datasource configuration and load-time weaving
- May conflict with built in JPA capabilities in a Java EE server
  - JNDI lookup generally more appropriate for Java EE
- Supports local and global transactions
- Example configuration shown below
  - It assumes the default *META-INF\persistence.xml*
  - Configure a different persistence unit location with the *persistenceXmlLocation* property

```
<beans>
  <bean id="personellEmf" class=
 "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="employeeDataSource"/>
  </bean>
</beans>
```

Container-Managed EntityManager

- *LocalContainerEntityManagerFactoryBean* provides a **transaction-scoped container-managed** entity manager
  - Generally, you inject an EM into a JPA-based DAO
  - A persistence context is associated with, and scoped to, the transaction's lifetime
  - Spring propagates the persistence context associated with the current transaction
  - When the transaction finalizes, the persistence context is flushed (if necessary) and then closed
  - This is the default behavior – you can specify differently (see notes)

- You use standard JPA annotations to inject the EM
  - Similar to how JPA is used in Java EE environment (where the EM can also be injected)

JEE: Obtaining an EntityManager From JNDI

- In Java EE, you can use Spring JNDI lookup for EMF lookup
  - Depends on the Java EE container for:
    - Autodetecting persistence units (i.e via *META-INF\persistence.xml* files in application jars)
    - Defining ENC locations (JNDI names) in the Java EE deployment descriptors using persistence-unit-ref
- Persistence unit deployment done by the Java EE server
  - As is *DataSource* configuration and integration of *EM* transactions with the JTA environment
  - Spring just retrieves the *EMF*, and injects it
    - May also manage transactions (via *JtaTransactionManager*)

```
<beans>
  <jee:jndi-lookup id="employeeEmf"
                   jndi-name="persistence/PersistenceUnit"/>
</beans>
```

LocalEntityManagerFactoryBean

- *LocalEntityManagerFactoryBean* creates an *EMF*
  - Suitable for JPA-only environments
  - Uses the JPA provider's autoscanning in a Java SE environment to detect persistent classes
  - Generally requires only the persistence unit name (from JPA's *persistence.xml*) be specified
  - Below, we show configuration with a persistence unit named "*events*"
  - Very little configuration available for this bean
    - No support for global transactions, nor support for integration with JPA providers that need to specify a JVM agent for weaving *

```
<bean id="eventEmf"
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="events"/>
</bean>
```

Additional Spring Configuration

- Additional configuration required for Spring/JPA integration
- **Annotation Scanning** – Spring post-processors must be configured to scan for standard JPA annotations
  - For injecting the EM
  - Done easily with the following element (as seen earlier):

```
<context:annotation-config/>
```

- **Vendor Adaptor**: Configures Spring/JPA-provider integration
  - Supports common configuration properties, including:
    - Should SQL traces be logged
    - The database platform
    - Should the schema be generated in the DB when the app starts
- The next slide shows both these configuration elements
  - Assuming a Hibernate-based JPA provider

Spring Configuration Example

- To use *LocalContainerEntityManagerFactoryBean*

```
<beans ...> <!-- See notes about namespaces -->

  <context:annotation-config/>

  <bean id="vendorAdapter" class=
      "org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="databasePlatform"
              value="org.hibernate.dialect.DerbyDialect"/>
    <property name="showSql" value="true"/>
    <property name="generateDdl" value="false"/>
  </bean>

  <bean id="personellEmf" class=
      "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="employeeDataSource"/>
    <property name="persistenceUnitName" value="javatunes"/>
    <property name="jpaVendorAdapter" ref="vendorAdapter"/>
  </bean>
</beans>
```

JPA Data Access Object

- A JPA-based DAO is shown below
  - Spring recognizes *@PersistenceContext*, and injects the EM
    - Thanks to the post-processors we configured (see notes)
  - If a TX and associated persistence context is present, the persistence context is associated with the injected EM
    - Otherwise a new persistence context is created
  - That's it – no Spring code in your DAO, and no need to manually propagate the EM – we'll see how useful this is shortly

```
public class JpaEmployeeDAO implements EmployeeDAO {
   @PersistenceContext
   private EntityManager em;   //get/set methods not shown

   public Employee searchById(Long id) {
      return  em.find(Employee.class,id);
   }
}
```

Extended Persistence Context

- Sometimes a persistence context should live longer than a transaction
  - In that case, you must create the EM yourself, and not inject it
  - You generally inject an entity manager factory (using *@PersistenceUnit*), and use it to create the EM
  - You are responsible for controlling the lifecycle of the EM in your code - e.g. closing it when you're done (see notes)
- The code fragment at bottom shows creation of the EM
  - You would close it when you're done with it

```
public class JpaEmployeeDAO implements EmployeeDAO {
   @PersistenceUnit
   private EntityManagerFactory emf;  //get/set methods not shown
   private EntityManager em;

   public JpaEmployeeDAO {
      em    =    emf.createEntityManager();
   }
}
```

Lab – Integrating Spring and JPA

Lab

- **Overview**: In this (optional) lab, we will work with the Spring/JPA integration, including
  - Setting up a database and configuring a datasource bean in spring
  - Configuring persistence annotation processing
  - Setting up a *LocalContainerEntityManagerFactoryBean* bean
  - Finishing a JPA-based DAO that injects an EM
- **Objectives**: To work with the Spring/JPA integration
- **Builds on previous labs**: None
  - You will work in a new project and directory for this lab
- **Approximate Time**: 35-45 minutes
- **Note**: We will be using the Derby database (Apache open source)
  - Originally Cloudscape, it was put into open source by IBM

Review of Lab Classes

Lab

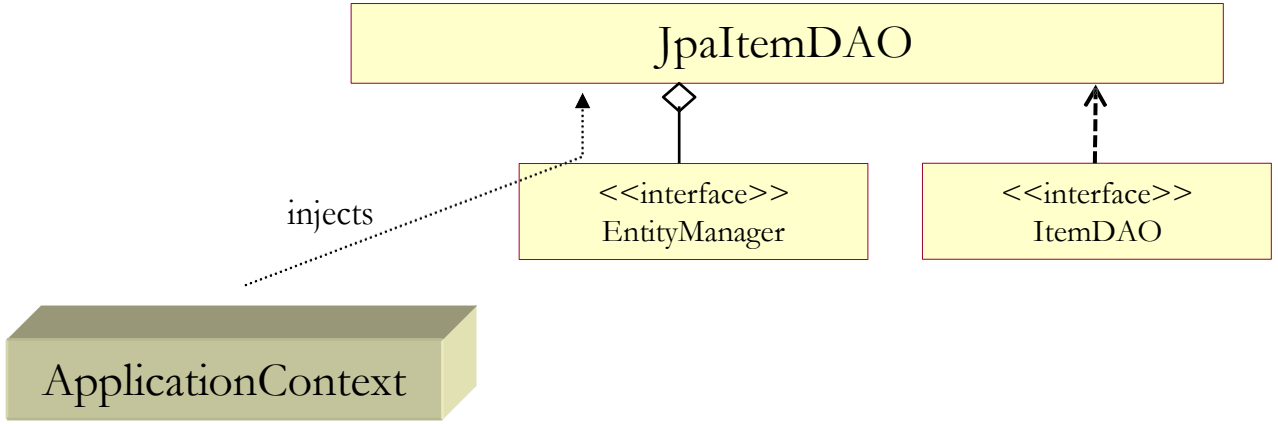
- You will work with the JavaTunes classes in this lab, including:
  - ***com.javatunes.util.MusicItem*** : JavaBean-style value class that represents a music item (e.g. CD, mp3 …) for sale
  - ***com.javatunes.catalog.ItemDAO*** : Interface that defines data access for JavaTunes items
  - ***com.javatunes.catalog.Catalog*** : Interface that defines the functionality of a catalog in JavaTunes
  - ***com.javatunes.catalog.JavaTunesCatalog*** : Concrete *Catalog* implementation

- You'll also need to finish the DAO implementation we supply:
  - ***com.javatunes.catalog.persist.JpaItemDAO*** : Concrete *ItemDAO* implementation that access a database using JPA
  - See the next slide

Our Lab Classes

Lab

- ***JpaItemDAO*** (Implements *ItemDAO*): JPA based MusicItem DAO – Has EM injected
  - Encapsulates JPA code, uses injected EM for persistence operations
- ***JavaTunes***: Top level test program
  - Uses *JavaTunesCatalog*
  - The catalog implementation uses a JPA-based DAO - but this is transparent, as it's injected as an *ItemDAO*



Lab Preparation

Lab

- The new root lab directory where you will do all your work is:

  C:\StudentWork\Spring\workspace\Lab04.1

Tasks to Perform

- Close all open files and projects
- Create a new Java project called Lab04.1 in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**
- Review the jars in the Spring library (in your Eclipse project)
  - Note these jars, which are required by Spring/JPA and Hibernate
    - spring-orm-n.n.n.RELEASE.jar: This is the Spring orm jar
    - hibernate-*.jar: Hibernate jars
    - slf4j-*.jar: slf4j support required for Hibernate logging
    - *derbyclient-n.n.n.n.jar*: This is the derby driver jar
    - Many other jars required by Hibernate/JPA

Setup/Create the Derby Database

Lab

Tasks to Perform

- Start the Derby database server
  - Execute ***dbStart.cmd*** (you can double-click on it)
  - This starts a standalone Derby server that can accept network connections to the database
  - You should see output like that below in the command window

```
Derby Server
Security manager installed using the Basic server security policy.
Apache Derby Network Server - 10.3.1.4 - (561794) started and ready to accept co
nnections on port 1527 at 2007-10-29 21:02:05.656 GMT
```

- Create the database
  - Execute ***dbCreate.cmd*** (you can double-click on it)
  - This creates the **JavaTunesDB** database which we'll use in the labs*
  - (you can ignore a DROP TABLE error, if you see it)

Using an SQL Command Tool

Lab

- **ij** is Derby's SQL command line tool
  - It allows us to create database objects and view and manipulate database data, without having to write code to do so
  - Most database packages provide such a tool

Tasks to Perform

- Execute ***dbSQL.cmd*** to run ij (you can double-click on it)
  - This command file is set up to connect to the correct DB
  - Using ij, you can browse the DB
  - Execute the query shown below to see the items in the DB

```
ij> select * from item;
    ... Lots of output ...
ij> exit;
```

Finish Spring Configuration

Lab

Tasks to Perform

- Open src***applicationContext.xml*** (The Spring configuration file)
  - Take note of the context namespace in the *<beans>* element
- Add in the ***<context:annotation-config/>*** element
  - This **enables annotation scanning**
- Finish the ***javatunesDataSource*** bean definition
  - Note that *javatunesDataSource* is a ***DriverManagerDataSource***
  - The Derby driver class is ***org.apache.derby.jdbc.ClientDriver***
  - The DB URL is ***jdbc:derby://localhost:1527/JavaTunesDB***
  - The username is ***guest***, the password is ***password***
- We will be using Hibernate as our JPA provider
  - Finish the ***vendorAdapter*** bean definition as follows:
  - Set the *class* to *org.springframework.orm.jpa.vendor.* ***HibernateJpaVendorAdapter***
  - This **configures the Hibernate provider for JPA**

Finish Spring Configuration

Lab

Tasks to Perform

- Finish the *javatunesEmf* bean definition, which configures our entity manager factory, as follows:
  - Set the *class* to: *org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean*
  - Set the *dataSource* property to *javatunesDataSource*
  - Set the *persistenceUnitName* property to *javatunes*
  - Set the *jpaVendorAdpater* to *vendorAdapter*
  - This finishes configuration for our EMF
- Review *src\META-INF\persistence.xml* also
  - The JPA persistence unit configuration
- That's it for configuration
  - You've configured a data source, the JPA provider (via the vendor adapter), and the EMF

Finish DAO and Test Program

Lab

Tasks to Perform

- Open ***JpaItemDAO.java*** (This is our DAO implementation)
  - Review the code – this is a DAO that encapsulates the JPA code
  - It uses an injected EM for its persistence functionality
  - You need to finish the code in this DAO as described below
- Add the ***@PersistenceContext*** annotation on the EM declaration so it will be injected
  - Notice that we don't have to initialize the EM – it will be injected into the DAO, and we simply use it within the class
  - Finish the *find(Long id)* method
    - It should simply call the entity managers find, then return the results it gets
- Note that the DAO is purposely kept very simple, with one method
  - We just include enough to demonstrate the Spring/JPA integration

Finish DAO and Test Program

Lab

Tasks to Perform

- Open ***JavaTunes.java*** for review
  - It already contains code that uses a catalog bean
  - The catalog bean now uses a JPA-based DAO, but this is transparent to the client
- **Run** the JavaTunes program and view the output
  - It should successfully find the item - done with the DAO and its injected JPA entity manager
  - That's it – you've successfully configured and used a JPA-based DAO that uses an injected EM

STOP

Review Questions

- What is a Data Access Object (DAO), and how is it used?
- How does Spring support Hibernate?
- How does Spring support JPA?

Lesson Summary

- A **DAO** encapsulates database access and insulates the rest of your application code from dealing with database details
  - It also gathers all the DB code in one place, making it easier to manage

- **Spring** provides a rich selection of types to help you create DAOs
  - It manages most of the low-level details of working with JDBC, such as acquiring and releasing connections
  - It also provides a fine-grained runtime exception hierarchy that is used instead of the checked *SQLException* from JDBC

Lesson Summary

- Spring has extensive support for using ORM technologies like **JPA** and **Hibernate**

- For Hibernate, Spring supports the configuration of Module Factory's
  - Which easily integrates with other Spring beans, such as datasources
  - It depends on contextual Module s for Module propagation

- For JPA, Spring supports the injection of entity managers or entity manager factories
  - It also has support for configuring the entity manager factory
  - This makes it extremely easy to integrate Spring and JPA – for the most part, the Spring support is transparent – the entity manager is injected

Lesson Summary

with the standard Java annotations

Spring Transaction Management
XML Configuration

Lesson Objectives

- Understand and configure Spring transaction managers
- Understand how declarative transaction management works
- Use Spring declarative transaction management with both annotations and XML configuration

LEARNINGPATTERNS

Spring Transaction Management

Spring Transaction Management

XML Configuration

Transaction (Tx) Overview

- **Transaction**: A collection of actions on the state of a system
  - Transformation of a system from one consistent state to another
  - The collection of actions must conform to the **ACID** properties below
- **A**tomicity: All of the work in a transaction is saved or none is saved
  - For example, all the actions in a bank transfer must occur
- **C**onsistency: A transaction is a correct transformation of the state
  - e.g., In a bank transfer the source account is debited the same amount the destination account is credited
- **I**solation: Even though transactions execute concurrently, it appears to each transaction that they occur sequentially
  - e.g., A bank transfer should be able to ignore any other transactions reading and writing the bank accounts
- **D**urability: Once a transaction completes successfully (commits), its changes to the state survive failures – like a server crash

Transaction Lifecycle

- A transaction is begun by the client or system
  - Thereafter, program operations are part of the transaction
  - All operations performed by other programs in the service of this program are part of the transaction
- **Committing** declares the operations to be complete & correct
  - Once the transaction commits, its effects are durable
- **Rolling back** undoes the operations
  - The system can cause a rollback if it detects some failure
- Transactions allow easy modularizing of a system
  - Each module is a transaction (or sub transaction)
  - If all goes well, the transaction commits and all the modules move to a new (durable) state
  - If something fails, the transaction is rolled back

Transaction Managers

- Spring **transaction managers** abstract the underlying transaction resource controlling transactions
  - Your code is not coupled to a particular transaction implementation
  - The transaction manager delegates to the underlying resource
    - They do NOT handle the transaction themselves
- Spring supplies a number of transaction managers
  - For many environments, including direct JDBC, Hibernate, Java EE, and JPA
- Each transaction manager requires appropriate resources
  - e.g., the JDBC transaction manager requires a *DataSource*,
  - The resources are generally configured in the Spring config

Configuring Transaction Managers

- In the examples below, we show two configuration examples
  - *DataSourceTransactionManager* - for a single JDBC *DataSource*
  - *HibernateTransactionManager* for a Hibernate *Module Factory*
  - That's all that's needed to configure them

```
<bean id="personnelDataSource"
 class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <!-- Detail not shown --> </bean>

<bean id="transactionManager" class=
    "org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="personnelDataSource"/>
</bean>
```

```
<bean id="personnel Module
   Factory" class="org.springframework.orm.hibernate3.Local Module
   FactoryBean">
   <!-- Detail not shown --> </bean>

<bean id="transactionManager" class=
    "org.springframework.orm.hibernate3.HibernateTransactionManager">
```

Configuring Transaction Managers

```
    <property  name=" Module
    Factory"  ref="  personnel Module
    Factory  "/>
</bean>
```

JTA Transaction Manager

- Spring provides ***JtaTransactionManager*** to support JTA transactions
  - JTA (Java Transaction API) is a standard part of the JEE spec
  - Available in all Java EE servers
- It's very easy to configure a *JtaTransactionManager*, as shown below
  - The *tranasctionManagerName* property allows configuration of a JNDI lookup name for the JEE server's transaction manager *

```
<bean   id="transactionManager"
  class="org.springframework.transaction.jta.JtaTransactionManager">
    <property   name="transactionManagerName"
                value="java:/TransactionManager"/>
</bean>
```

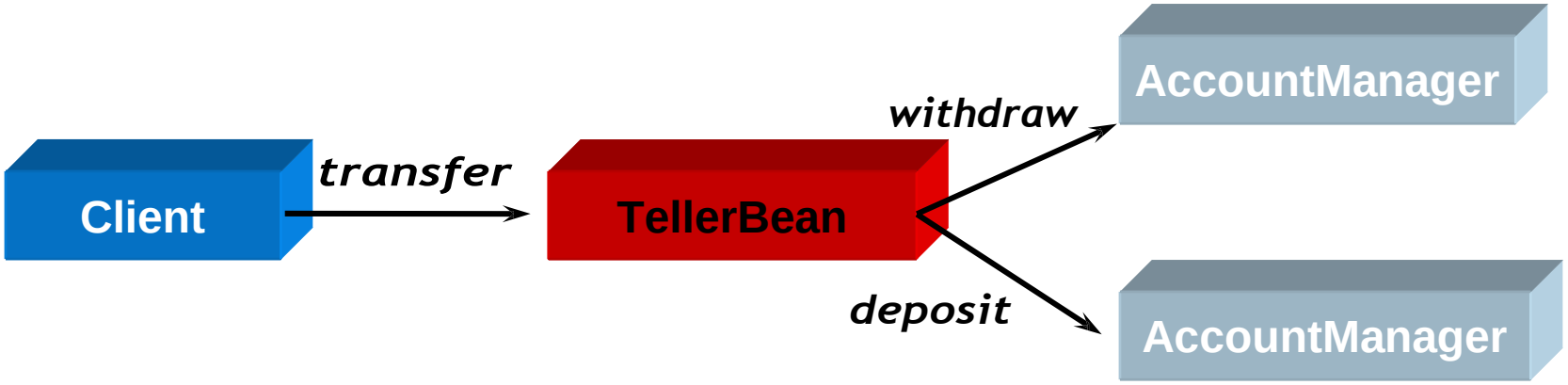
Spring Declarative Transaction Management

- Spring allows you to **declare** a bean's transactional behavior
  - Rather than coding to an API to explicitly control the transaction
  - You define transaction **boundaries** across which a transaction **propagates**, and where transactions can be affected
  - We'll cover the behavior first, then look at adding it to your code
- Based on **Spring AOP**
  - Spring intercepts the method calls and adds in TX behavior
- TX declarations often done with annotations
  - Which requires the configuration below (from the tx namespace)

```
<tx:annotation-driven
                    transaction-manager="transactionManager"/>
```

Spring Transactional Scope

- The **scope** of a transaction is comprised of all the beans in the TX
  - A transaction **propagates** from a bean to other beans that it invokes on
  - The transaction also propagates to resources used from a bean (e.g. a database connection)
  - In the simple banking example below, if a TX is started in *TellerBean*, it will propagate to the *AccountManager* beans
  - If one of the beans accesses a database, the TX will propagate to it



Transaction Propagation

- The transaction context an invocation runs is determined by:
  - **Transaction attributes**: e.g. REQUIRED
  - **The current transactional state**: Is a transaction active
  - There are transaction attributes for **propagation**, **isolation**, and **timeout** of transactions, and for **read-only** transactions
  - We'll cover the TX attributes next
- Depending on the above, the container may
  - **Start a transaction**, if one is not active
  - **Propagate an existing transaction** from one bean invocation to another
  - **Suspend** an existing transaction (to prevent its propagation)
  - This ensures all beans participate in a transaction appropriately

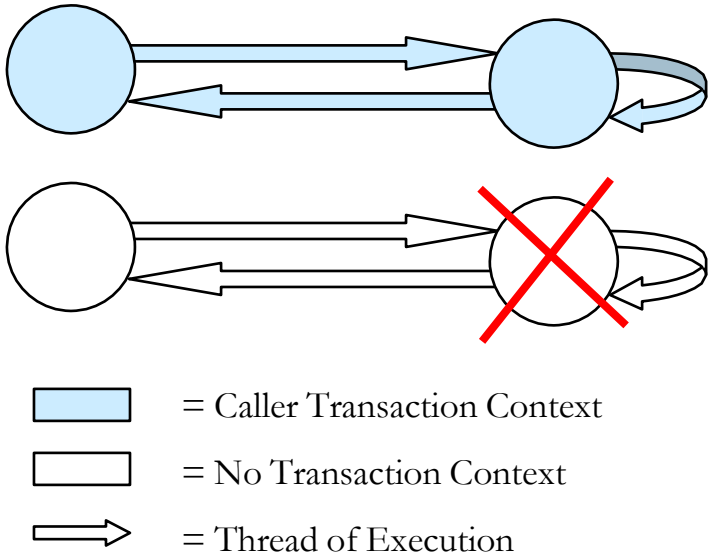
Transaction Attributes for Propagation

- Spring supports the following transaction attributes controlling TX propagation
  - *MANDATORY*
  - *NESTED*
  - *NEVER*
  - *NOT_SUPPORTED*
  - *REQUIRED*
  - *REQUIRES_NEW*
  - *SUPPORTS*

- These can be specified at both a bean & method level
  - **Bean level**: The default for all methods in a bean
  - **Method level**: Behavior for the specific method
    - Overrides any bean level declaration

MANDATORY

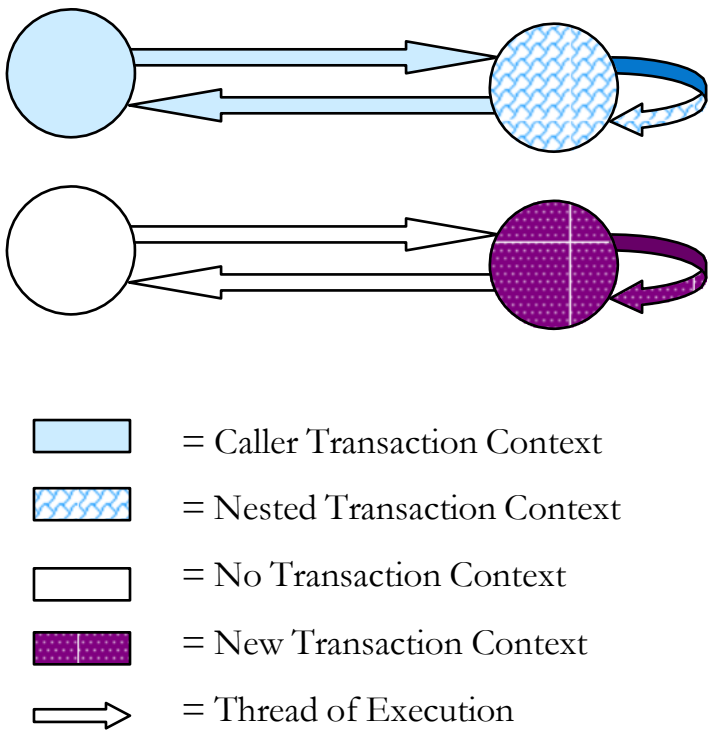
- Invocation **must** occur within the scope of the caller's TX
  - If the client tries to invoke the bean without a TX context, the container throws the *TransactionRequired* exception



- a) Incoming TX propagated by invocation on second bean
- b) No incoming TX, ERROR, exception thrown !

NESTED

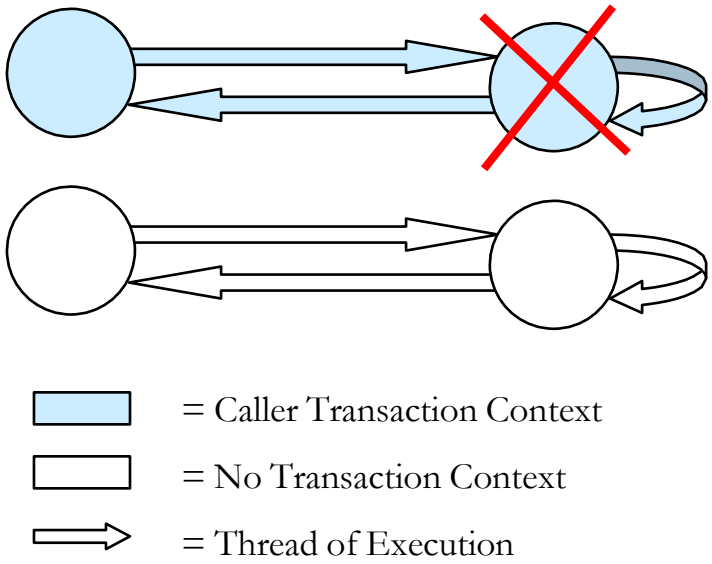
- Invocation always occurs in a TX context
  - If a client is associated with a TX, a nested TX is started
  - If a client is not associated with a TX, a new TX is started



- a) Nested TX started, invocation occurs in nested TX
- b) No incoming TX, so invocation occurs in new TX

NEVER

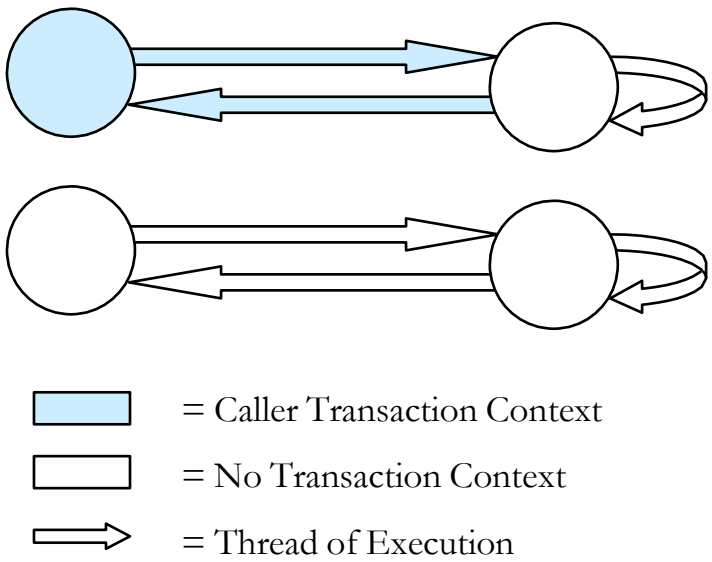
- Caller is **required** to call the bean method **outside** the scope of a TX
  - Container invokes bean without a TX context



- a) Incoming TX, ERROR, exception thrown
- b) No incoming TX, and no TX in invocation

NOT_SUPPORTED

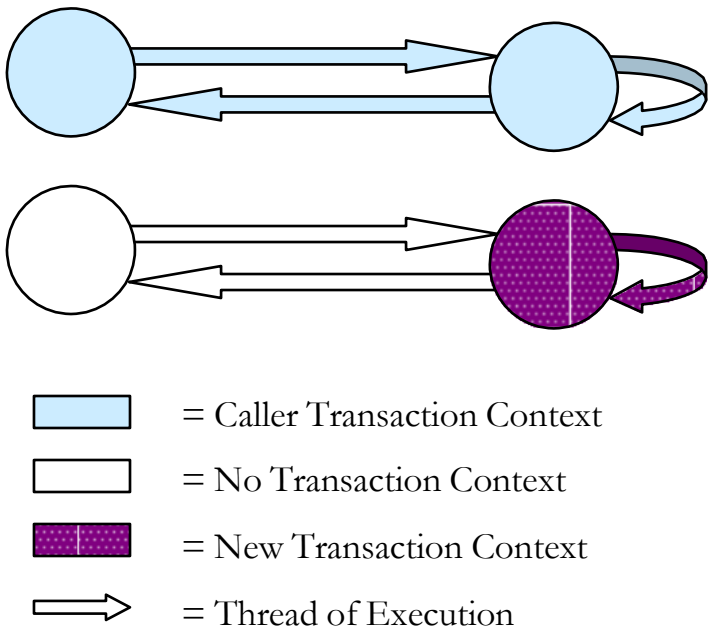
- Invocation should **not** be executed within a TX
  - A container should always invoke the bean outside a TX
  - If a client calls within a TX, the TX is suspended, the method is executed, and then the TX is resumed
  - The suspended TX context is not passed to resources invoked from the object



- a) Incoming TX suspended in invocation on second bean
- b) No incoming TX, and no TX in invocation

REQUIRED

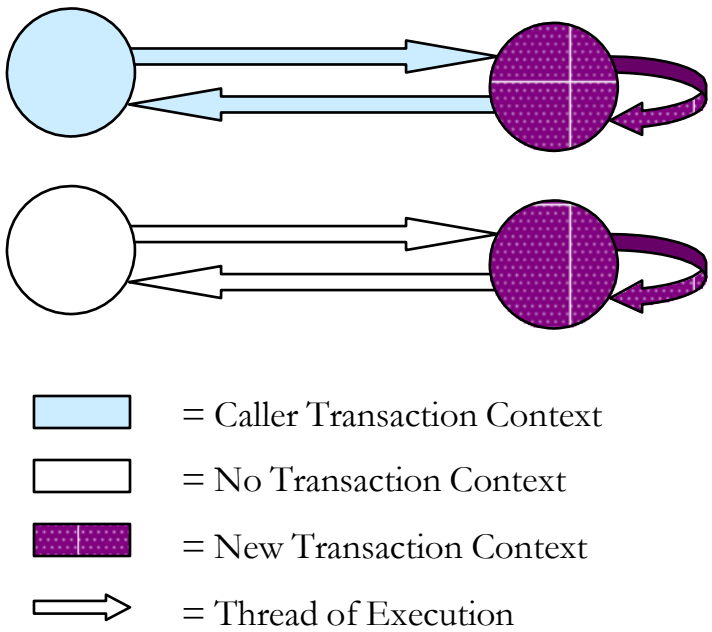
- If caller is in a TX, the bean method is invoked in that TX context
- If caller is not in a TX, bean method is invoked in a new TX
  - The container attempts to commit the TX when the method call completes



- a) Incoming TX propagated by invocation on second bean
- b) No incoming TX, so invocation occurs in new TX

REQUIRES_NEW

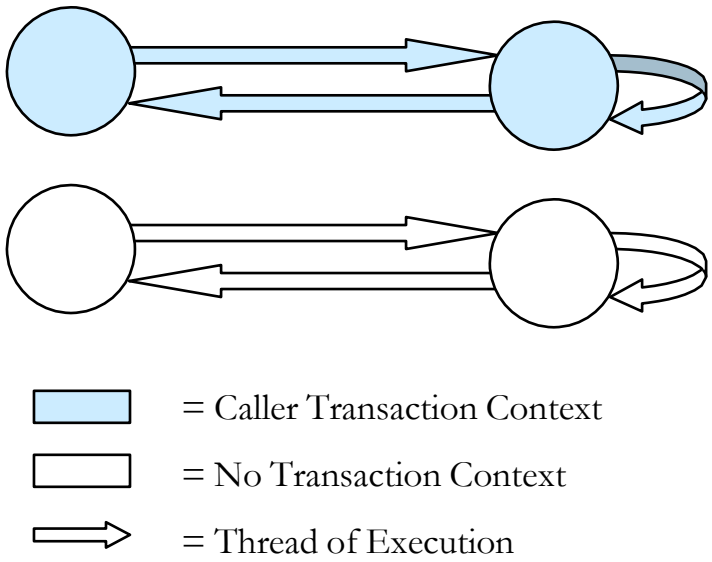
- Invocation always occurs in a new TX context
  - The container starts a new TX before invoking a method, and attempts to commit it when the method returns
  - If a client is associated with a TX, it is suspended before the new TX is started, and resumed when it completes



- a) Incoming TX suspended, invocation occurs in new TX
- b) No incoming TX, so invocation occurs in new TX

SUPPORTS

- The bean method is invoked in the **caller's** TX scope
  - If the caller has a TX context, it is passed in
  - If the caller doesn't have one, then there is no TX context

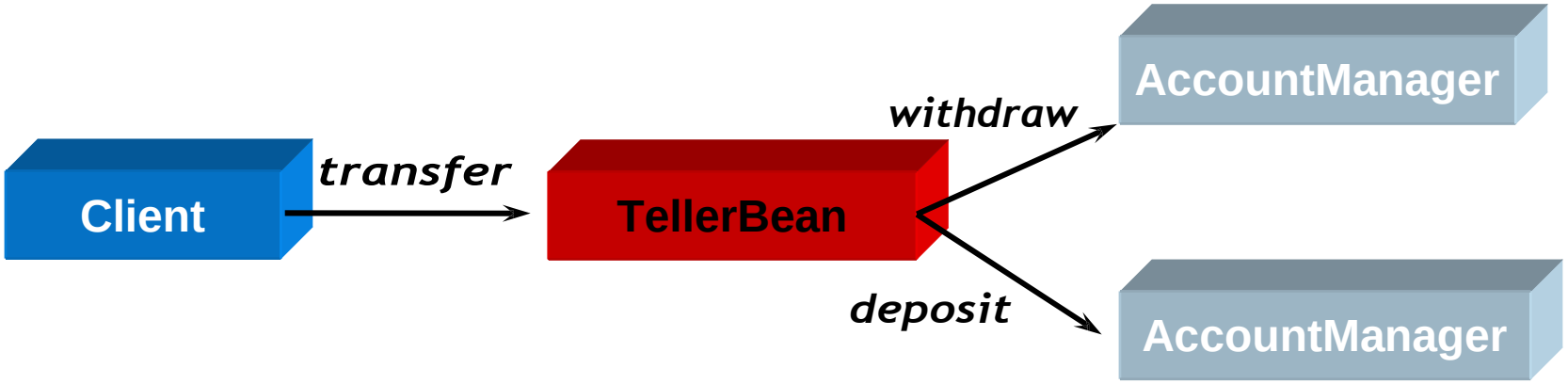


- a) Incoming TX propagated by invocation on second bean
- b) No incoming TX, and no TX in invocation

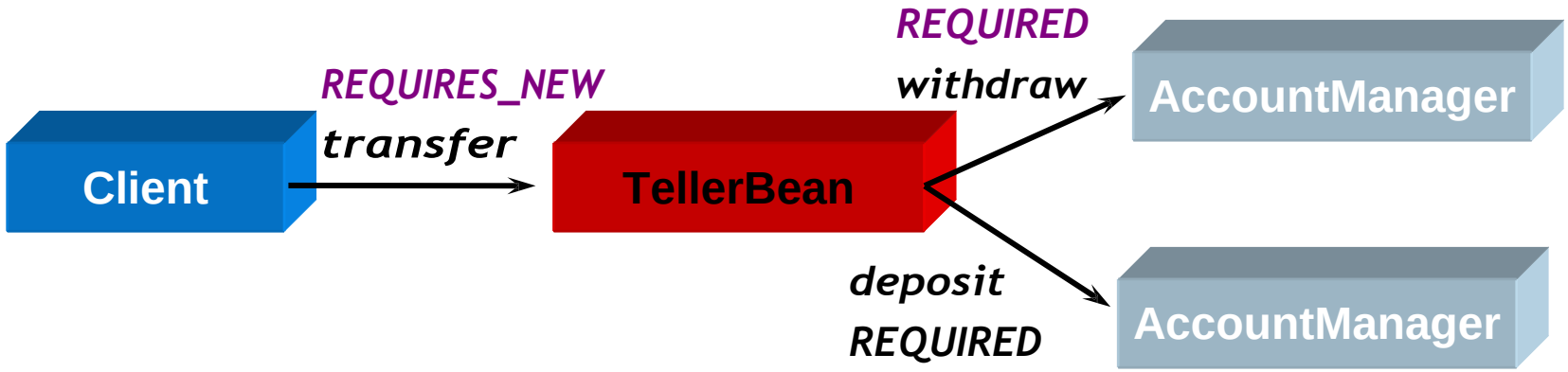
Transaction Example

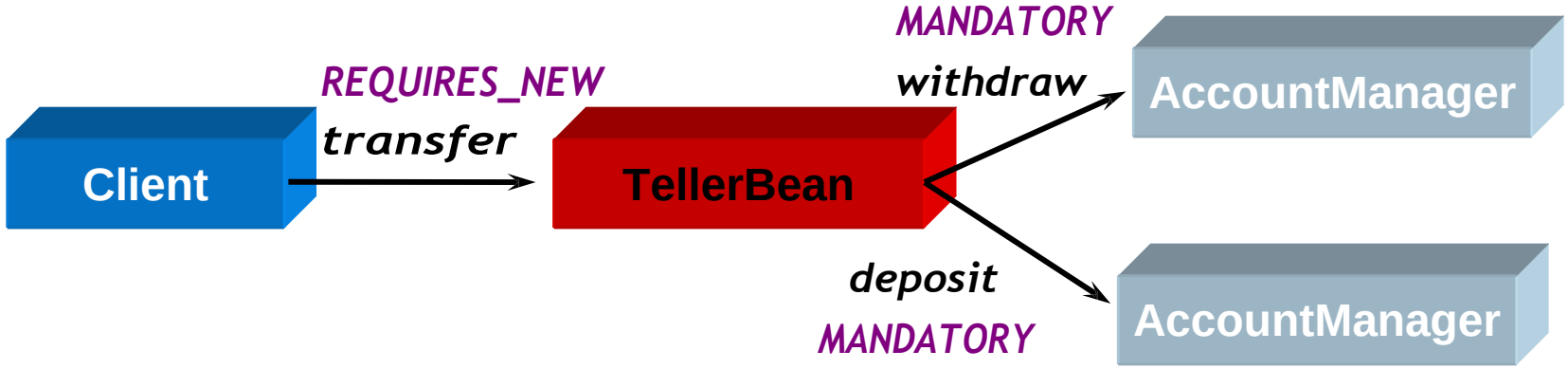
- A transfer involves two account objects
  - You withdraw from one, deposit to another
  - How do you do this in one transaction?

- One way to do this is to create a teller Module bean that has a *transfer* method calling *withdraw* on the "from" account and *deposit* on the "to" account

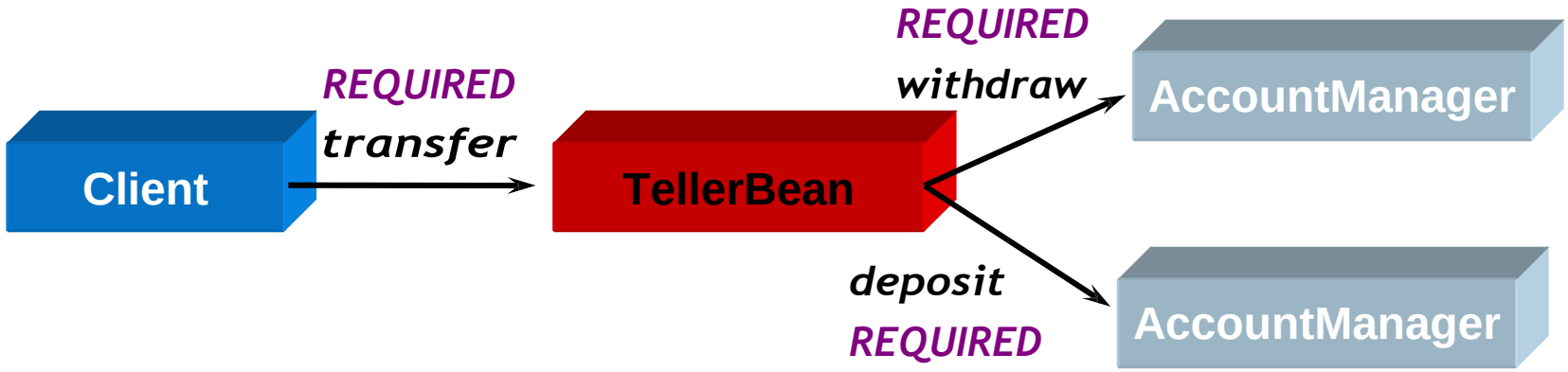


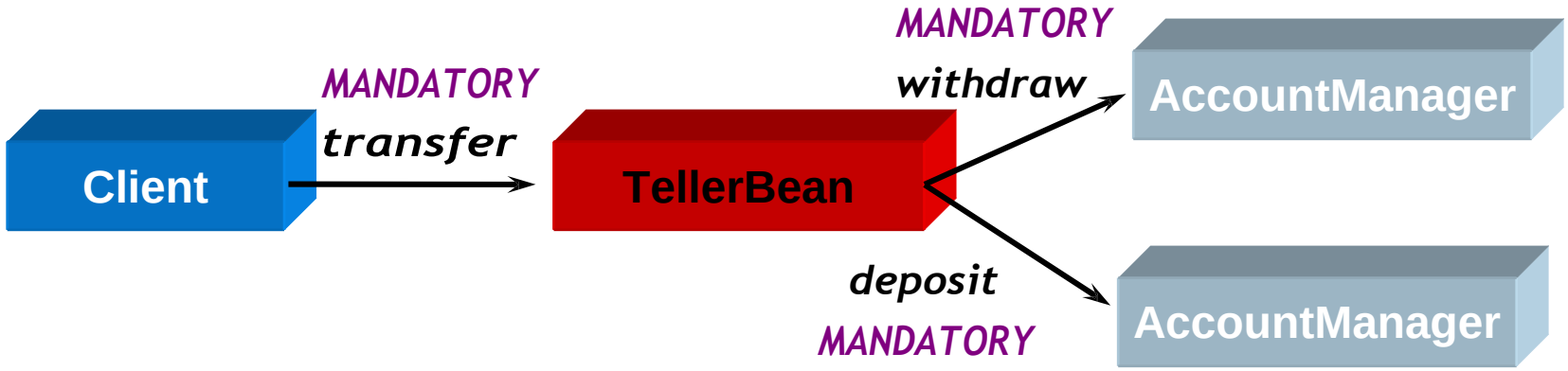
Transaction Attributes – Some Choices





Transaction Attributes - Some Choices





Specifying Transaction Attributes

- Transaction attributes can be declared with Spring annotations
  - In package: *org.springframework.**transaction.annotation***
- **@Transactional** is the main annotation
  - It describes transaction attributes on a method or class
- Two enumerations also defined in the package
  - ***Isolation*** defines supported transaction isolation levels
  - ***Propagation*** defines transaction propagation behaviors
- Declarative transactions makes programming much simpler
  - No need to code to a potentially complex API
  - Removes need for transactional code mixed with business code
- Numerous facets to this control
  - We'll look at TX propagation first

Specifying Transaction Attributes

- Below, *JdbcEmployeeDAO* methods are given a default TX propagation of ***SUPPORTS***
  - We override this in *create()* with a propagation of ***REQUIRED***
  - *create()* **always** runs within a transaction
  - The search methods run in the caller's TX scope
  - Declaring TX behavior is **easy** !

```
import    org.springframework.transaction.annotation.Propagation;
import    org.springframework.transaction.annotation.Transactional;

@Transactional(propagation=Propagation.SUPPORTS)
public class JdbcEmployeeDAO extends JdbcDaoSupport
                                        implements EmployeeDAO {
  public Employee searchById(int id) { /* Details not shown */ }
  public Collection searchByName(String name) { /* … */ }

  @Transactional(propagation=Propagation.REQUIRED)
  public int create(Employee emp) { /* Details not shown */ }
}
```

Additional Transactional Attributes

- Other transactional attributes of *@Transactional* include:
- ***Isolation isolation***: Degree of isolation this TX has from the work of other transactions
  - DEFAULT, READ_COMMITTED, READ_UNCOMMITTED, REPEATABLE_READ, SERIALIZABLE
- ***int timeout***: how long (seconds) the tx may run before timing out
- **boolean readOnly**: true if the transaction is read-only.
  - A read-only TX does not modify any data
  - Sometimes a useful optimization (such as when using Hibernate)
- Below, we set *readOnly* status & *timeout* for *searchByName()*

```
@Transactional(readOnly=true, timeout=10)
public Collection searchByName(String name) { /* … */ }
```

Transactional Attributes Guidelines

- Generally, TX annotations are placed on a **concrete class**
  - Not on an interface, which may not work in all situations *
  - Only **public methods** work with the TX annotations
    - Assuming the proxy-based implementation (details covered later)
- Your annotated classes should **implement an interface**
  - This allows the use of JDK dynamic proxies (see notes)
  - This is fairly standard when using Spring anyway
- Proxy mode (the default) is limited
  - Self-invocations will not be wrapped in transactions, even if annotated
  - We'll cover this later

Rolling Back and Exceptions

- By default, Spring marks a TX for rollback if an unchecked (e.g. runtime) exception is thrown
  - This is the recommended way to trigger a rollback
  - The container will catch an unhandled exception, and mark the active TX for rollback
  - Checked exceptions do not trigger a rollback by default

- Refine rollback behavior with the following *@Transactional* elements
  - ***Class[] noRollbackFor***: Classes that should NOT trigger a rollback
  - ***String[] noRollbackForClassName***: Names of classes that should NOT trigger a rollback
  - ***Class[] rollbackFor***: Classes that SHOULD trigger a tx rollback
  - ***String[] rollbackForClassName***: Names of classes that SHOULD trigger a rollback

Lab 5.1 – Spring Transactions

Lab – Spring Transactions

Lab

- **Overview**: In this lab, we will work with the Spring transaction capabilities, including
  - Configuring annotation driven transactions in the Spring config
  - Adding transactions to a bean
  - Working with different transactional behavior
  - The lab uses Spring's JDBC support to provide an environment where we can work with the transaction support
- **Objectives**: To work with the Spring transaction support
- **Builds on previous labs**: None
  - You will work in a new project and directory for this lab
- **Approximate Time**: 35-45 minutes

Lab Preparation

Lab

- The new root lab directory where you will do all your work is:

  C:\StudentWork\Spring\workspace\Lab05.1

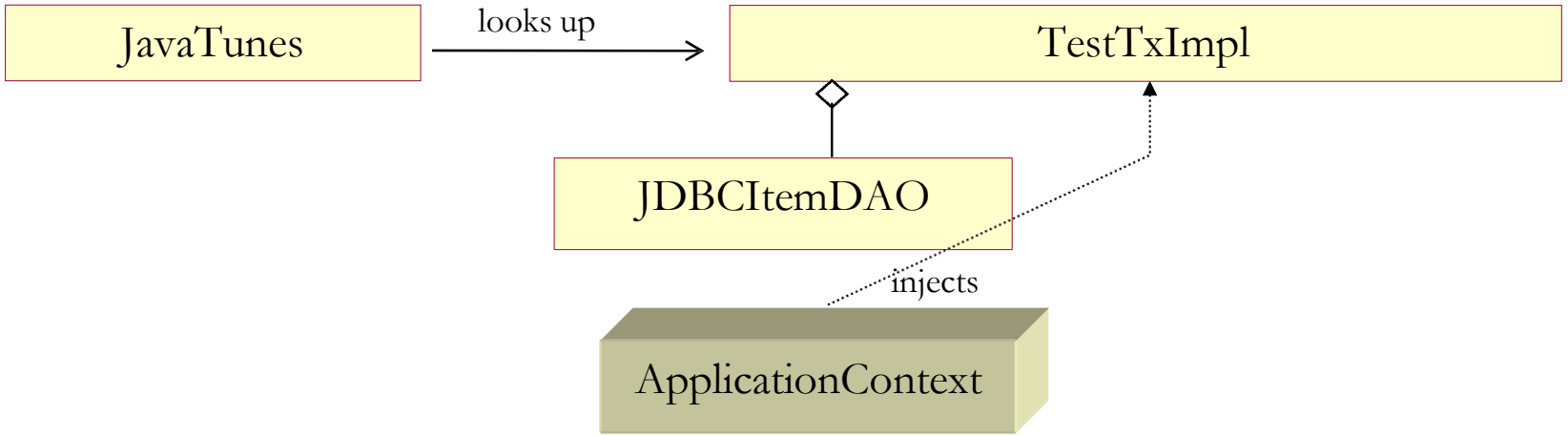
Tasks to Perform

- Close all open files and projects
- Create a new Java project called Lab05.1 in the workspace
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**
- Make sure that the database is running
  - See Lab 4.1 if you need details on starting the DB

Our Lab Classes

Lab

- ***JDBCItemDAO*** (implements *ItemDAO*): JBDC-based DAO
  - DAOs have new method *blowUp()* which throws a runtime exception (we'll use it to roll back a transaction)
- ***TestTxImpl*** (implements *TestTx*): Test bean that has a *MusicItemDAO* injected
  - One method, *runTest()*, that first calls *create()*, then calls *blowUp()* on the DAO
- ***JavaTunes***: Top level driver program



Finish Spring Configuration

Lab

Tasks to Perform

- Open ***applicationContext.xml*** (The Spring configuration file)
  - Take note of the tx namespace in the *<beans>* element
- Add in a **DataSource transaction manager bean** (see notes)
  - The *dataSource* property should refer to the *javatunesDataSource* bean that is already present in the config file
  - This bean integrates the spring transactions with your DataSource
- Add in the ***tx:annotation-driven*** element (see notes)
  - The transaction-manager attribute should refer to the transaction manager you just created
  - This element enables annotation driven transactions
- Note the declaration of the *JdbcItemDAO* bean we supply

Persistence Context Propagation

Lab

Tasks to Perform

- Open ***TestTxImpl.java*** and review the *runTest()* method
  - It simply uses its injected *MusicItemDAO* bean to call *create()*, then *blowUp()*
- Open ***JavaTunes.java*** for review – note that it looks up a TestTx bean, and calls *runTest()* on it
  - It then searches for the item created in *runTest()* (via *create()*)
- **Run** the program, and look at the console output
  - You should see that a new music item was created
  - *create()*'s transaction finalizes before *blowUp()* is called

```
Created container
Exception thrown in TextTxImpl: Blowup
*** searchByKeyword for 'Transaction' found the following: ***

com.javatunes.util.MusicItem: id=22: num=CD599 title=Transaction Blues artist=We are Commi
*** searchByKeyword - Transaction finished ***
```

Transaction Rollback

Lab

Tasks to Perform

- In ***TestTxImpl.java*** annotate *runTest()* with ***@Transactional*** and the proper propagation so a transaction is **REQUIRED**
- **Run** the program again, and look at the console output
  - There should not be any new items created – no matter how many times you run the program
  - We're running in the context of the transaction started in *runTest()*
    - The call to *blowUp()* rolls this tx back
    - The call to *create()* is running in the same tx, so it rolls back also

```
Created container
Exception thrown in TextTxImpl: Blowup
*** searchByKeyword for 'Transaction' found the following: ***

com.javatunes.util.MusicItem: id=22: num=CD599 title=Transaction Blues artist=We are Commi
*** searchByKeyword - Transaction finished ***
```

Creating a New Transaction

Lab

Tasks to Perform

- In ***JdbcItemDAO.java*** annotate *create()* with ***@Transactional*** with the proper propagation to require a NEW transaction
- **Run** the program again, and look at the console output
  - You should now see that an additional item was created
  - Since the call to *create()* runs in its own transaction, the call to *blowUp()*, which rolls back its transaction, has no effect on *create()*
- You can see how easy it is to control TX behavior with Spring

```
Exception thrown in TextTxImpl: Blowup
*** searchByKeyword for 'Transaction' found the following: ***

com.javatunes.util.MusicItem: id=22: num=CD599 title=Transaction Blues artist=We are Commit
com.javatunes.util.MusicItem: id=26: num=CD599 title=Transaction Blues artist=We are Commit
*** searchByKeyword - Transaction finished ***
```

[Optional] Look at TX Flow

Lab

[Optional] Tasks to Perform

- If you have time, use *TransactionSynchronizationManager* to get information about the TX flow
  - It's in *org.springframework.transaction.support*
  - We already include some output using this in the starter code
  - Use ***getCurrentTransactionName()*** in different places in your code to get the current transaction name, and output it to the console
  - This can help you see what TX is active - which is helpful in getting an understanding, and can also be helpful in debugging
  - This class is generally not used in user code, but can be useful when learning or debugging

```
System.out.println("TextTxImpl.runTest: " +
    TransactionSynchronizationManager.getCurrentTransactionName());
```

Spring Proxies and Direct Invocation

- Spring transactions are based on Spring AOP
  - Spring AOP, by default, works via proxies
  - The method invocation is intercepted, and the TX is added in
  - This affects TX behavior, as described below
- You must access an object **through a proxy** for transactions to work
  - i.e. - Via a reference originally looked up from the application context
  - Otherwise, the container can't add in the TX behavior
- If you are in a class, and call another method on the same class, **the invocation won't include any new TX behavior**
  - Because the invocation is through the *this* reference, not through a proxy

Example – Invoking Directly

- What happens when *empDAO.createAll()* is called
  - *create()* called multiple times, and it's set to REQUIRES_NEW

```
import    org.springframework.transaction.annotation.Propagation;
import    org.springframework.transaction.annotation.Transactional;

public class JdbcEmployeeDAO implements EmployeeDAO {

  @Transactional(propagation=Propagation.REQUIRED)
  public void createAll(Collection<Employee> employees) {
    for (Employee emp : employees) {
      create(emp); // Call create() directly
    }
  }
  @Transactional(propagation=Propagation.REQUIRES_NEW)
  public int create(Employee emp) { /* Details not shown */ }
}
```

```
EmployeeDAO empDAO = (EmployeeDAO)ctx.getBean("employeeDAO");
Collection<Employee> emps = // Initialized in some manner …
empDAO.createAll(emps);   // What happens here
```

Example – Invoking Directly

- *create()* has a *REQUIRES_NEW* TX attribute in the example
  - In the example, **that TX is NOT activated**
  - The call to *create()* goes through the ***this*** reference
  - That is **not a proxy**, so proxy-based behavior doesn't run
  - It may be possible to refactor your code to solve this
    - Cumbersome, not always possible
  - Better solution - configure Spring AOP / TX so proxies are not used
- You can configure Spring to use **load-time weaving**, which doesn't have this issue, via:

  <context:load-time-weaver/>

  - Replaces the proxy-based behavior

Load-Time Weaving

- Load-Time weaving (LTW) weaves aspects into class files as they are loaded into the JVM
  - So the aspects (including TX) become part of the class file
  - Not dependent on Spring proxies
  - Based on AspectJ capabilities that are integrated into Spring
- The configuration below configures load-time weaving
  - ***mode="aspectj"*** configures Spring to weave classes at load-time with Spring's AspectJ transaction aspect
  - ***aspectj-weaving="on"***: Specifies AspectJ weaving be used, even if there are no AspectJ configuration files
    - *META-INF\aop.xml* files (see notes)

```
<tx:annotation-driven  mode="aspectj"/>

<context:load-time-weaver  aspectj-weaving="on"/>
```

Spring's Load-Time Weaving

- *context:load-time-weaver* registers a number of LTW-related beans
  - Enables LTW by utilizing the existing support in many common app servers, with no other changes
  - Including JBoss, GlassFish, WebSphere, Weblogic, and others
- Using it in a Java SE environment requires adding support for class instrumentation (to weave the byte code)
  - Can be done by starting the JVM with a Java agent provided with Spring, as shown below
  - Not applicable to application server environments

```
-javaagent:C:\spring-framework-3.2.2.RELEASE\libs\
                    spring-instrument-3.2.2.RELEASE.jar
```

[Optional] Lab 5.2: Load-Time Weaving

Lab: Load-Time Weaving

Lab

- **Overview**: In this lab, we will enable load-time weaving for our program
  - Using the Java agent approach, since we're in a Java SE environment
  - We'll then test some code that will not work in a proxy-based approach (since it uses self-invocation)
- **Objectives**: Configure and use Spring's load-time weaving for TX support
  - Test load-time weaving
- **Builds on previous labs**: 5.1
- **Approximate Time**: 20-30 minutes

Test Self-Invocation and TX Behavior

Lab

Tasks to Perform

- Open *JdbcItemDAO* for editing
  - Uncomment the *runTest()* method
  - This method calls *create()*, then *blowUp()* using self-invocation
- Open *ItemDAO* for editing
  - Uncomment the *runTest()* method
- Open JavaTunes for editing
  - Comment out the block of code calling *test.runTest()*
  - Uncomment the block of code calling *dao.runTest()*
  - Your program now calls the code that uses self-invocation
- **Run** your program again
  - You should not see any new items created
  - Even though create() is annotated with REQUIRES_NEW, the call is not being wrapped in a transaction, since it's called via self-invocation
  - Self-invocation does not work in a proxy-based environment

Add Load-Time Weaving

Lab

Tasks to Perform

- Open src***applicationContext.xml*** (The Spring configuration file)
  - In the ***tx:annotation-driven*** element, add the following attribute to turn on AspectJ load-time weaving

    mode="aspectj"
  - Add in the following element to configure the load-time weaving

    <context:load-time-weaver aspectj-weaving="on"/>
- Open the Run Configuration for your program
  - Menu item **Run | Run Configurations**, then choose the JavaTunes run configuration for this program (It will have a project of Lab05.1)
  - Click on the arguments tab, and add the following in the VM arguments (with no newlines or spaces included)
    - Note: The **hyphen (-) before javaagent must be included**

-javaagent:C:\spring-framework-3.2.2.RELEASE\libs\spring-instrument-3.2.2.RELEASE.jar

  - Adjust accordingly for different versions or install locations

Test Self-Invocation and TX Behavior

Lab

Tasks to Perform

- **Run** your program again
  - You **should** see new items created
  - Self-invocation will work with load-time weaving
- You may also that your program starts/runs more slowly
  - Instrumenting the classes, and running the instrumented classes has a noticeable amount of overhead

STOP

[Optional] XML Configuration

Spring Transaction Management

XML Configuration

Specifying Transactions Using XML

- Spring supports TX specification with XML
  - Via tags in *tx* and *aop* namespaces
  - We'll go (very) briefly into some of the details here
- Below, is an example TX configuration using XML
  - ***<tx:advice>***: Defines the transactional semantics of methods
  - ***<tx:attributes>***: Contains transactional attributes
  - ***<tx:method>***: Specifies the TX attributes of the named method(s)
    - Below, methods with names that start with *create* or *search* have **REQUIRED** TX propagation
    - The search methods are read-only

```
<tx:advice id="defaultDaoAdvice">
  <tx:attributes>
    <tx:method name="create*" propagation="REQUIRED"/>
    <tx:method name="search*" propagation="REQUIRED"
               read-only="true"/>
  </tx:attributes>
</tx:advice>
```

Spring Transactions and AOP

- Spring's XML configuration is based on **AOP** configuration
  - AOP (Aspect Oriented Programming) **encapsulates crosscutting concerns** in one place called **aspects**
  - Covered in depth in the Module on AOP

- Key AOP concepts:
  - **Aspect**: A module holding cross-cutting concerns
  - **Advice**: Action taken by an aspect (its functionality)
  - **Joinpoint**: Point in program execution (e.g. method call), where advice may run
  - **Pointcut**: Specification matching joinpoints. Advice runs at any joinpoint matching its pointcut
  - **Advised (target) object**: An object being advised by an aspect
  - **Weaving**: The combining of aspects with the rest of your app

Defining a Pointcut

- **Joinpoint**: Point in program execution (e.g. method call), where advice may run
  - For TX purposes, they are the places where TX attributes may be applied
- A **pointcut** is a specification that matches joinpoints
  - Basically, it specifies locations in your code (the joinpoints)
  - Defined by a ***pointcut*** element, and an AspectJ expression specifying the joinpoint
  - The id becomes the name of the pointcut
  - The AspectJ expression specifies which joinpoints are of interest
  - Below, we specify a pointcut called ***anyTeach()***, that will match "**any execution of a method called teach**"

```
<aop:config>
  <aop:pointcut id="anyTeach" expression="execution(* teach(..))"/>
</aop:config>
```

Linking Advice With Pointcuts

- TX advice must be associated with a pointcut to run
- AOP pointcuts are used, as shown below
  - ***defaultDAOOperation*** defines a (broad) pointcut applying to any method of any type with DAO in the type's name
    - e.g. *JdbcItemDAO*
  - *<aop:advisor>* associates the pointcut to *defaultDaoAdvice*
  - *JdbcEmployeeDAO*'s *create* and *search** methods (seen earlier) will now run with *defaultDaoAdvice*'s TX attributes
    - Since the class has DAO in its name

```
<aop:config>
  <aop:pointcut id="defaultDAOOperation"
                expression="within(*..*DAO*)"/>
  <aop:advisor  pointcut-ref="defaultDAOOperation"
                advice-ref="defaultDaoAdvice"/>
</aop:config>
```

<tx:method> Attributes

- *<tx:method>* includes the following attributes:
  - ***name***: The method names to associate the tx attributes with
    - Wildcards (*) can be used in method names - e.g. on*Event, create*
  - Attributes corresponding to their *@Transactional* counterparts
    - ***propagation***: The transaction propagation behavior
      - default = REQUIRED
    - ***isolation***: The transaction isolation level, default = DEFAULT
    - ***timeout***: transaction timeout value (sec.) default = -1
    - ***read-only***: Is transaction read-only? default = false
    - ***rollback-for***: Exceptions that will trigger rollback; Value is comma-delimited list of classnames
    - ***no-rollback-for***: Exceptions that will NOT trigger rollback; Value is comma-delimited list of classnames

Why Use XML Configuration

- XML configuration of TX behavior is very powerful
  - It's easy to create a pointcut matching many methods in many classes
  - Allows you to broadly apply TX behavior in a very concise way
  - Annotating all those classes is potentially much more work
- However, it is also **fragile and unpredictable**
  - Changing the name of a class (e.g. from *MyDAO* to *MyDao*) might change the TX behavior, with no indication
    - Other than incorrect behavior !
- You must understand AOP in general, its capabilities, and its issues to use this properly
  - The AOP Module
    in this course contains more information

Review Questions

- What are Spring transaction managers?
- What is declarative transaction management, and how does it work?
- How do you use declarative transaction management in Spring?

Lesson Summary

- Spring **abstracts your program** from the underlying transaction resource via the use of transaction managers
  - Spring has transaction managers for most major environments
  - Resources: JDBC connection, JTA transaction manager, etc.
- Spring supports **declarative specification** of TX behavior
  - Rather than explicitly coding to an API
  - The container propagates a TX from one bean to another based on TX attributes on the bean classes
  - Some attributes include: propagation, isolation, timeout, read-only, rollback-for, no-rollback-for
- **Annotations** or **XML configuration** can be used to declare the TX attributes

LEARNINGPATTERNS

Module 6: Web Applications with Spring MVC

Integration with Java EE

Spring MVC Basics

Forms and Command Objects

Lesson Objectives

- Integrate the Spring container with regular Java Web Applications (first section)

- Understand what Spring MVC is, its goals, and its architecture
  - Learn how to configure the Spring MVC *DispatcherServlet*
  - Use Command Controllers to handler requests
  - Use View Resolvers to map logical names to actual resources (e.g. JSP)
  - Use Spring MVC's support for forms including command classes and form controllers
  - Understand how Spring uses Handler Mappings to map URLs to controllers

© 2007-13 LearningPatterns Inc. All rights reserved.

LEARNINGPATTERNS

Integration with Java EE

Integration with Java EE

Spring MVC Basics

Forms and Command Objects

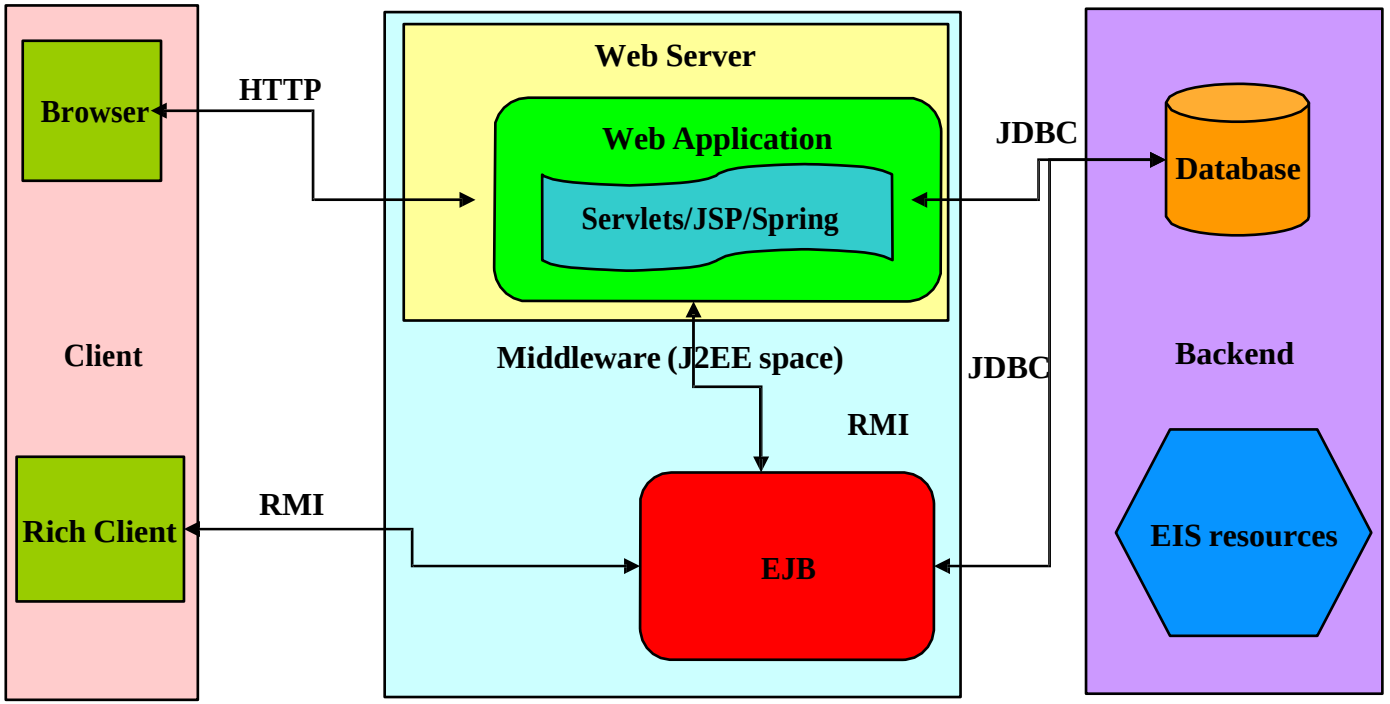
Additional Details

Spring and Java EE

- **Java EE** (**Enterprise Edition**): Standard Java architecture for building scalable, distributed, reliable, Enterprise apps
  - Enterprise Edition basically means "Server Side" Java
- Spring can complement or replace many parts of Java EE
  - Often useful to integrate existing web apps with Spring
  - Fairly easy to use **Spring beans and DI** with Java EE Web apps
  - We'll look at this first - it is used for Spring MVC integration also
- We'll look at **Spring MVC** after this
  - Powerful web framework to build complex web apps more easily

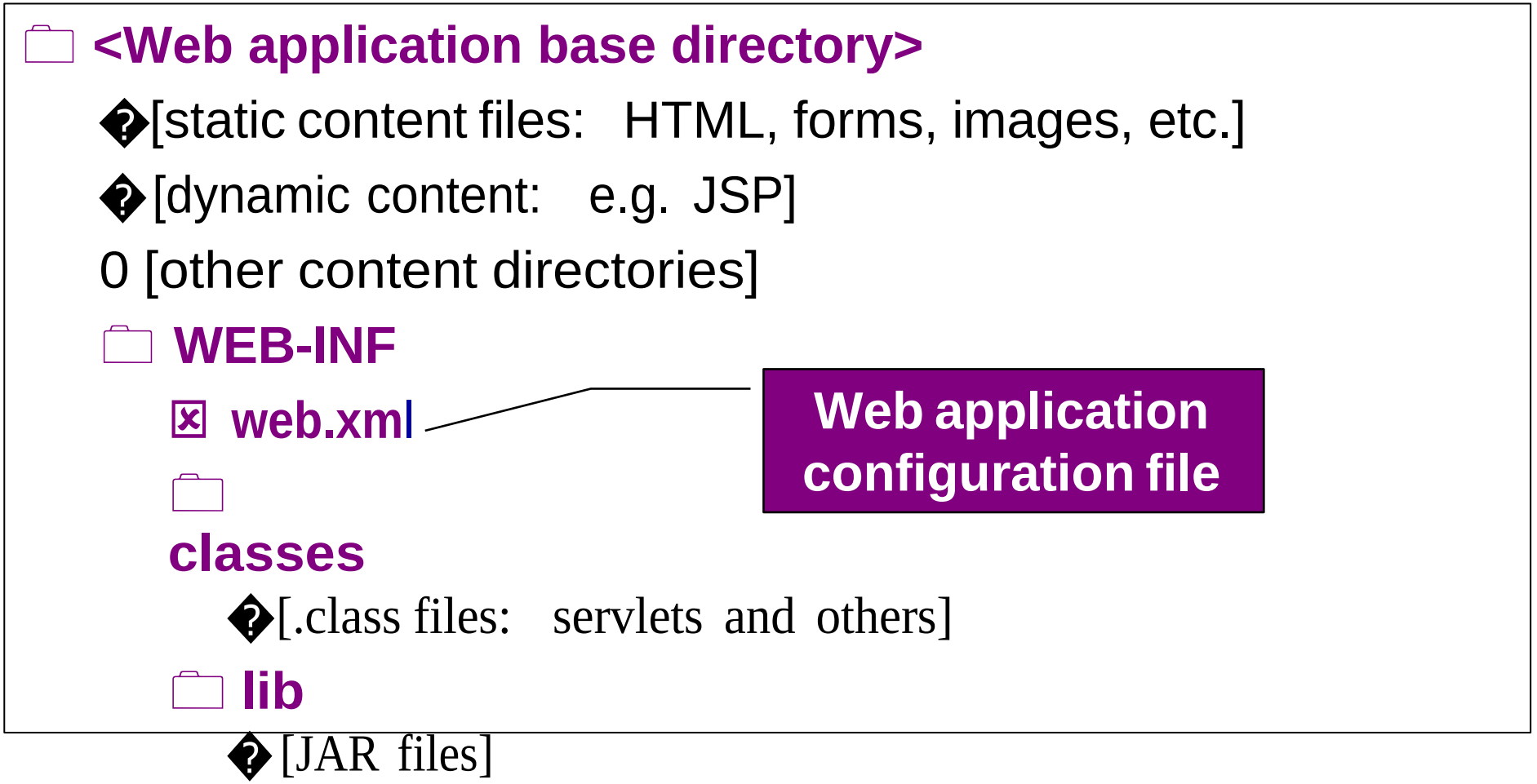
Overview of Java EE Web Applications

- Standard structure defined in Java EE
  - A collection of **Servlets**, **JavaServer Pages** (JSP), & other files
  - Most frequently collected inside of a Web Archive (WAR) file
    - The standard Java EE web app packaging



Web Application Structure

- Java web apps are organized in the standard structure below



Web Application Components

- **Servlets**: Java components that run on the server
  - Servlets run **in response to client requests**
  - Defined by extending class ***javax.servlet.HttpServlet*** and overriding ***doGet()/doPost()***

- **JSP**: JavaServer Page - contains dynamic and static content
  - **Template Data** consists of regular text, HTML, XML ...
  - **Dynamic Data** is generated anew for each request - e.g from custom tags in the JSP

- **web.xml**: Deployment descriptor for web app
  - Contains configuration information for the web app
  - For example, the servlets and their mappings

ApplicationContext and Java Web Apps

- Spring's ***WebApplicationContext*** provides integration with the Java Web container
  - This is a context tailored for Web app use
- Spring's ***ContextLoaderListener*** makes loading the context easy
  - Automatically loads the context when the web app is deployed
  - You can define the Spring configuration file locations using a *<context-param>* element in *web.xml*
- ***WebApplicationContextUtils*** class is used to access the context in your web app code
  - Via the static method ***getWebApplicationContext()***
  - If using standard Java Web apps
  - The following slides shows examples of doing this

Configuring ContextLoaderListener

- The *web.xml* below configures the *ContextLoaderListener*
  - It specifies *applicationContext.xml* and *inMemoryItemDAO.xml* as config files via *<context-param>*

```
<web-app ... >

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

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

</web-app>
```

Using the Application Context

- Below, we retrieve the context via *WebApplicationContextUtils*
  - This is the context loaded by the *ContextLoaderListener*
  - Bean look up is the same as in non-web related code
  - The example assumes the same configuration and beans as in our JavaTunes labs earlier in the course
- You can see - it's very easy to use Spring with Web apps
  - For other frameworks, like Spring MVC, integration is even easier *

```
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  =  ctx.getBean("javaTunesCatalog",  Catalog.class);
}
```

LEARNINGPATTERNS

Lab 6.1 – Spring and the Web

Lab 6.1 – Spring and the Web

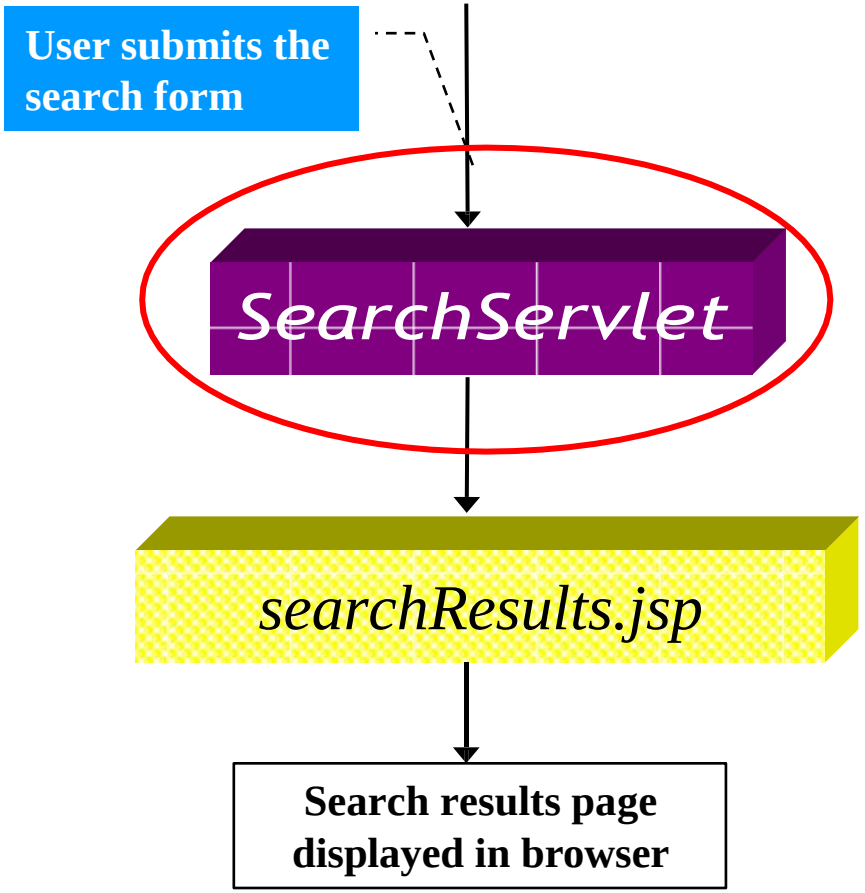
Lab

- **Overview**: In this lab, we will integrate the Spring container with a regular Java Web application
  - We will need to set up our Web server (Tomcat)
  - We will then configure the *ContextLoaderListener* in *web.xml*, and use the application context in a servlet
    - We'll look up a Spring bean (a catalog) that's used in a JSP page
- **Objectives**:
  - Configure *ContextLoaderListener*
  - Use an application context from a servlet
- **Builds on previous labs**: none
- **Approximate Time**: 30-40 minutes

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



Eclipse and Web Projects

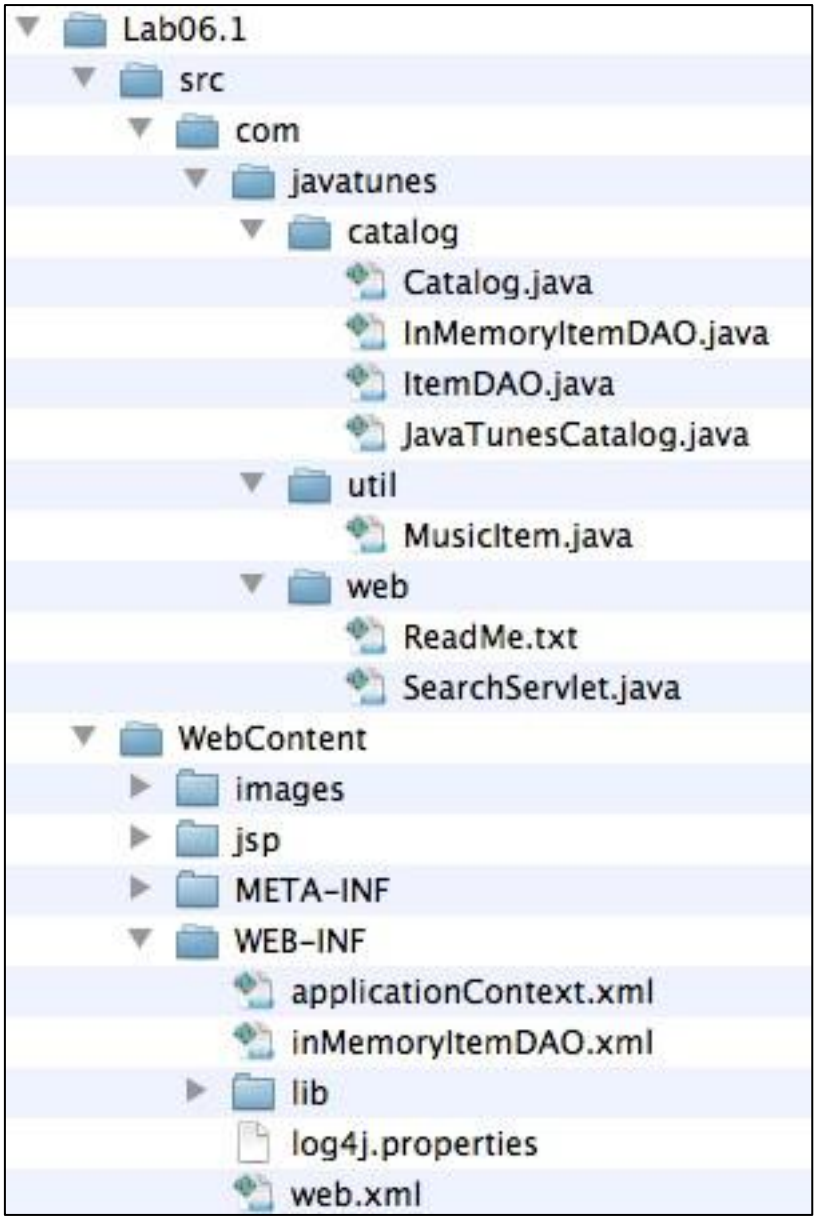
Lab

- **Dynamic Web Projects** used for Java Web apps in Eclipse
  - These contain servlets, JSP pages, and static resources (HTML)
  - Web-specific project properties can be establish when creating the project
  - *web.xml* has a custom editor for ease of use
- Projects are organized in the following folders
  - **src**: Contains all Java source files
  - **WebContent**: Contains all Web resources
  - **WebContent\WEB-INF**: Same as Java EE WEB-INF
    - Also contains Spring configuration files, other config files
  - This is different from a standard Java EE Web app
    - When deployed, a standard WAR is build

Lab Directory Structure

Lab

- The labs in this section are structured as follows:
  - **LabNN** : Directory for Lab NN
  - **LabNN/src/** : Java source
  - **LabNN/WebContent/** : jsp, HTML
  - **LabNN/WEB-INF/**: web.xml, other config files
  - **LabNN/WEB-INF/lib**: jar files needed by the Web app



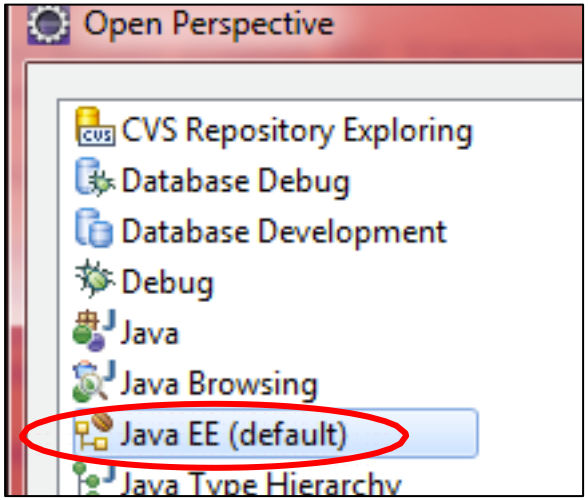
Lab Preparation

Lab

- The root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab06.1
  - This is a new lab directory which already exists in your workspace

Tasks to Perform

- **Close** all open files and projects
- Open the Java EE perspective, by clicking the Perspective icon at the top right of the Workbench, and select **Java EE**
  - This perspective is well suited for these labs



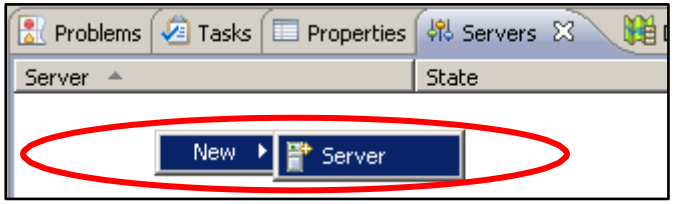
Creating a Server

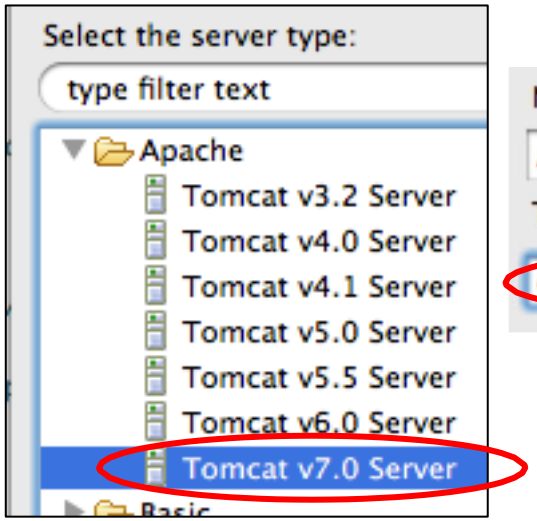
Lab

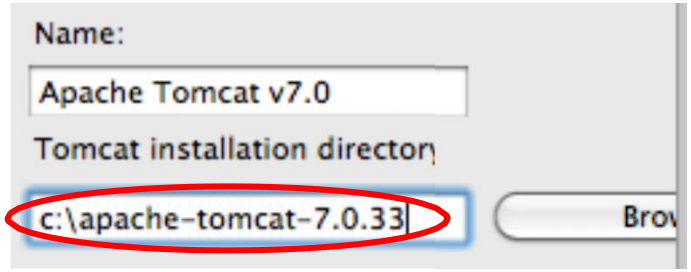
- We will use Tomcat to run our Web applications - first we need to create a server in Eclipse *

Tasks to Perform

1. Go to the Servers view, right click, and select **New | Server**
2. In the next dialog, select **Apache | Tomcat V7.0** * and click **Next**
3. In the next dialog, browse to your **Tomcat install directory**, click **OK**, and then **Finish** *







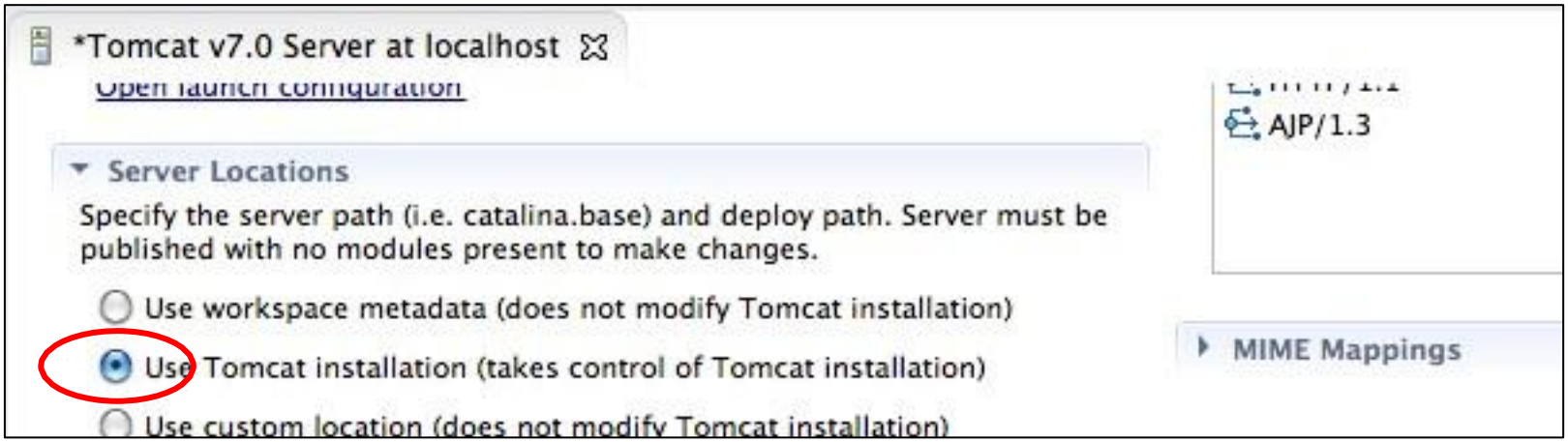
Configure Server

Lab

- We will reconfigure the deploy location for our server
  - This will help prevent Eclipse deploy issues we've seen (see notes)

Tasks to Perform

- In **Servers view**, double click on the Tomcat server
  - The server configuration should open in the editor pane
  - Check the "Use Tomcat installation" choice (see below)
  - Save the configuration

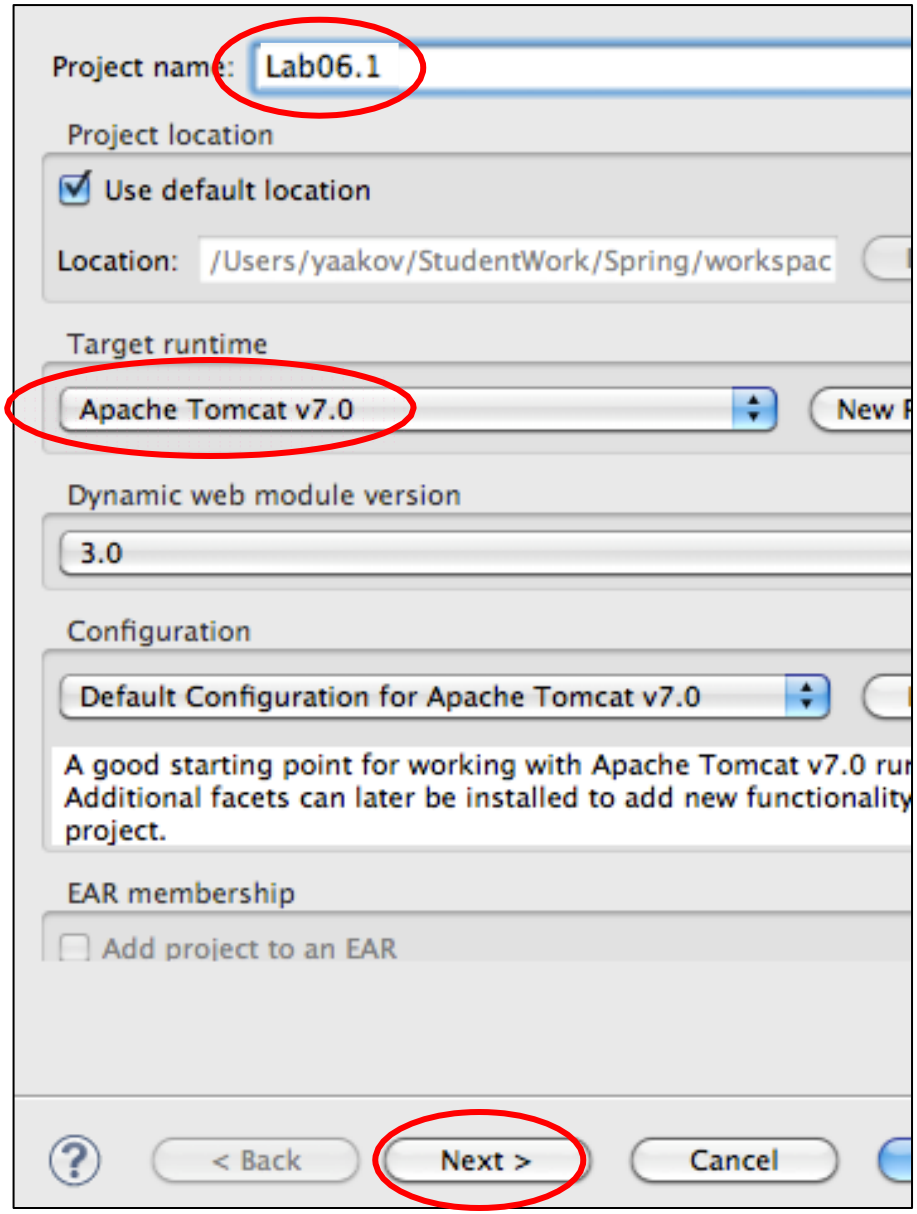


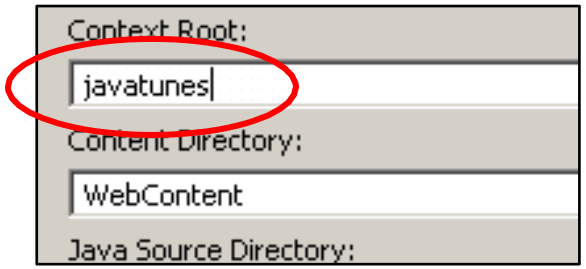
Create a Web Project

Lab

Tasks to Perform

- Create a new Dynamic Web project (as at right)
  - **File | New | Dynamic Web Project** *
  - Call it **Lab06.1 ***
  - Make sure the **Tomcat runtime** is selected and Click **Next**
- In the next dialog, change the Context Root to **javatunes** (**all lower case** - as below)
  - Click **Finish**





Configure Build Path

Lab

- We need to add the Spring library (including the web support) to our project classpath to allow the Web app to compile/run properly

Tasks to Perform

+Right click on the project in Project Explorer, select **Build Path | Configure Build Path**, add the Spring library as for other projects

+Next, click **Deployment Assembly** in the left pane (for configuring Web app deployment dependencies)

- Click **Apply** if prompted to save modifications
- Click **Add | Java Build Path Entries** click **Next**, select the **Spring** library, click **Finish**, then **OK**

+Review the jars in the Spring library (in your Eclipse project)

- Note the jars that support the Spring/Web integration
  - e.g. jars from Spring like spring-web-3.2.2.RELEASE.jar, and the spring-web-servlet jar
  - There are others in the dependencies we supply - e.g. the JSTL jar

Configure ContextLoaderListener

Lab

Tasks to Perform

- Note: All Web content (JSP files, WEB-INF folder, …) appears in the Project Explorer pane under the **WebContent** folder
- Open up *WEB-INF\web.xml* for editing
  - Locate the *<context-param>* element and add in the values for the elements shown below:
  - *<param-name>*: *contextConfigLocation*
  - *<param-value>*:
    /WEB-INF/applicationContext.xml
    /WEB-INF/inMemoryItemDAO.xml
  - Locate the *<listener>* element and add in the following value for the *<listener-class>* element:

org.springframework.web.context.ContextLoaderListener

Finish SearchServlet

Lab

Tasks to Perform

- Open *com.javatunes.web.**SearchSevlet*** for editing
  - Notice the Spring imports for the web context classes
  - Find ***doPost()***, & initialize the *WebApplicationContext* by calling the static ***WebApplicationContextUtils*** method ***getRequiredWebApplicationContext()*** (See manual slides)
  - Pass in a servlet context retrieved via *getServletContext()*
- Review the rest of the *doPost()* code
  - It uses the context to look up a *Catalog*
  - It uses the catalog to do keyword based search
    - Using a keyword from a search form a web page
  - Search results are put on the request under the name "results"
  - The request is forwarded to */jsp/searchResults.jsp* for display *

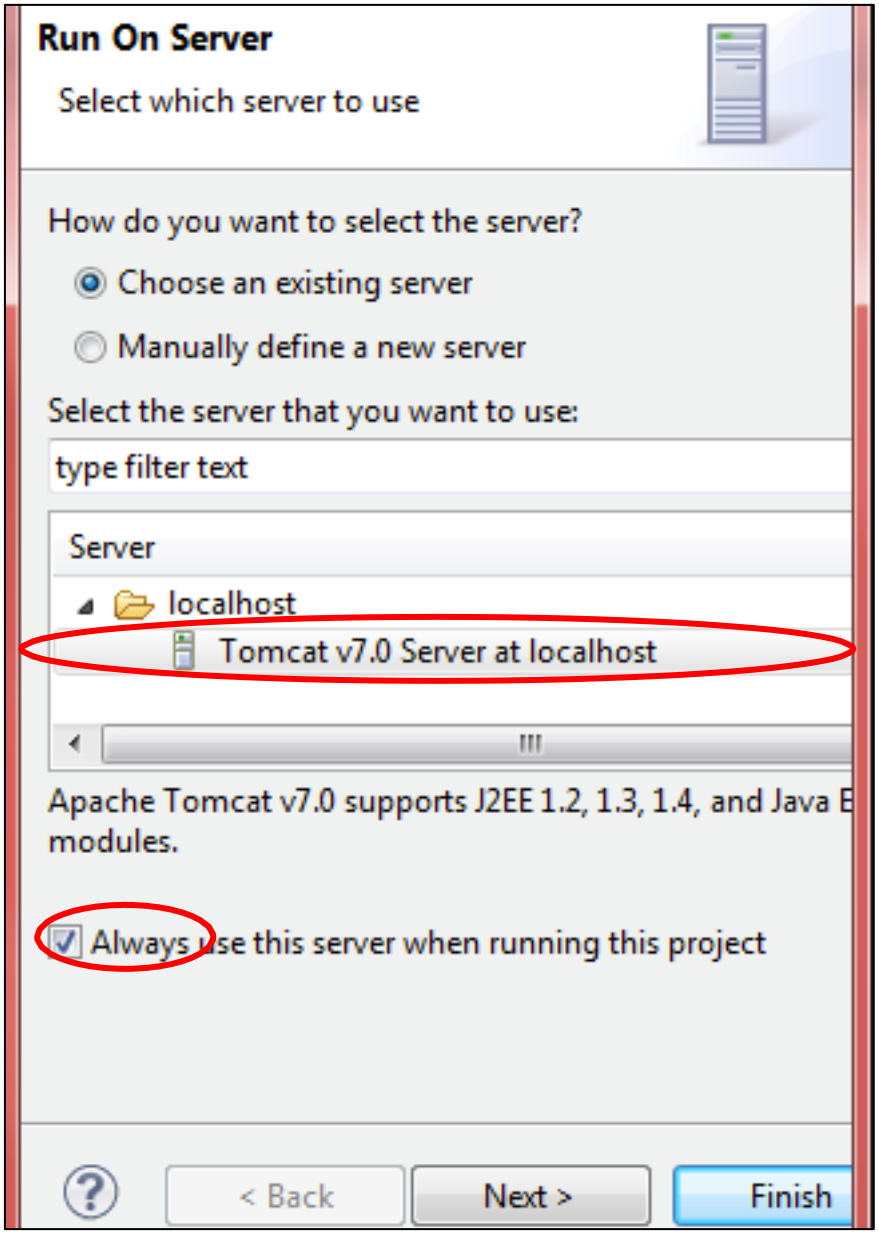
Deploying an Application

Lab

Tasks to Perform

- To deploy to the server, right click on the Lab06.1 project, and select **Run As | Run on Server**
- In the next dialog box select your existing **Tomcat** server
- Also select **Set server as project default**, as we will always use this server
- Click **Finish**

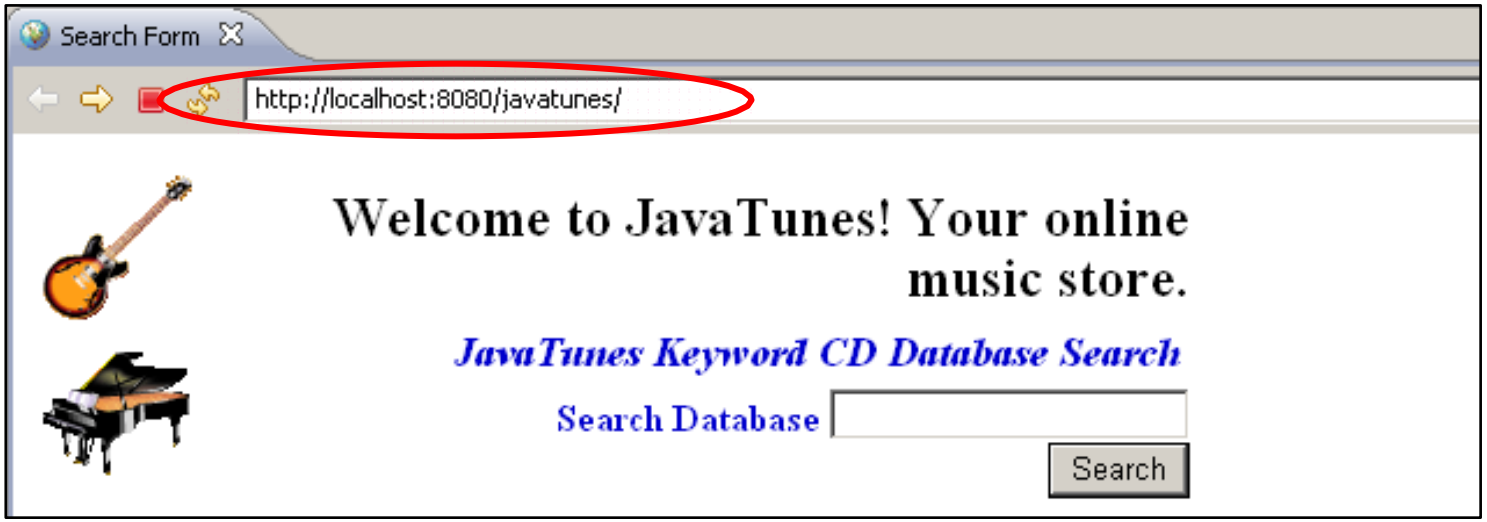
- Note that running a Web app on the server will automatically start the server



Viewing the Web Application

Lab

- Eclipse should open a browser for you onto the Web app
  - At: *http://localhost:8080/javatunes*
- Note: Eclipse uses the system default browser, often IE, and IE **can be misleading** as it caches pages, and it's hard to clear the cache
  - If you ever feel your having browser issues, open an external browser viewing the same URL (see notes about setting the browser in Eclipse)



Test Your Application

Lab

Tasks to Perform

- The **javatunes** web app, displays a search form - so type in the letter "a" to search on and submit the form
  - The *SearchServlet* will be invoked, use your Spring beans to do the search, and then forward to a JSP to display the results
  - That's it - you're using Spring in a Web application

Server Startup

Lab

- You can open the **Console** view to see output from the server startup and web app deployment
  - This can be useful to look for exception stack traces
  - Note that server startup may take some time, especially the first time you start the server
  - You can also look at the server status in the **Servers** view

Markers | Properties | Servers | Data Source Explorer | Snippets | Console

```
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre7\bin\javaw.exe (Dec 19, 2012 2:04:45 PM)
Dec 19, 2012 2:04:48 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Dec 19, 2012 2:04:48 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2216 ms
2012-12-19 14:05:24,561 DEBUG [org.springframework.beans.factory.support.DefaultListableBeanFactory] -
JavaTunesCatalog:findByKeyword - nn
maxSearchResults = 20
Dec 19, 2012 2:05:24 PM org.apache.jasper.compiler.TldLocationsCache tldScanJar
```

STOP

LEARNINGPATTERNS

Spring MVC Basics

Integration with Java EE

Spring MVC Basics

Forms and Command Objects

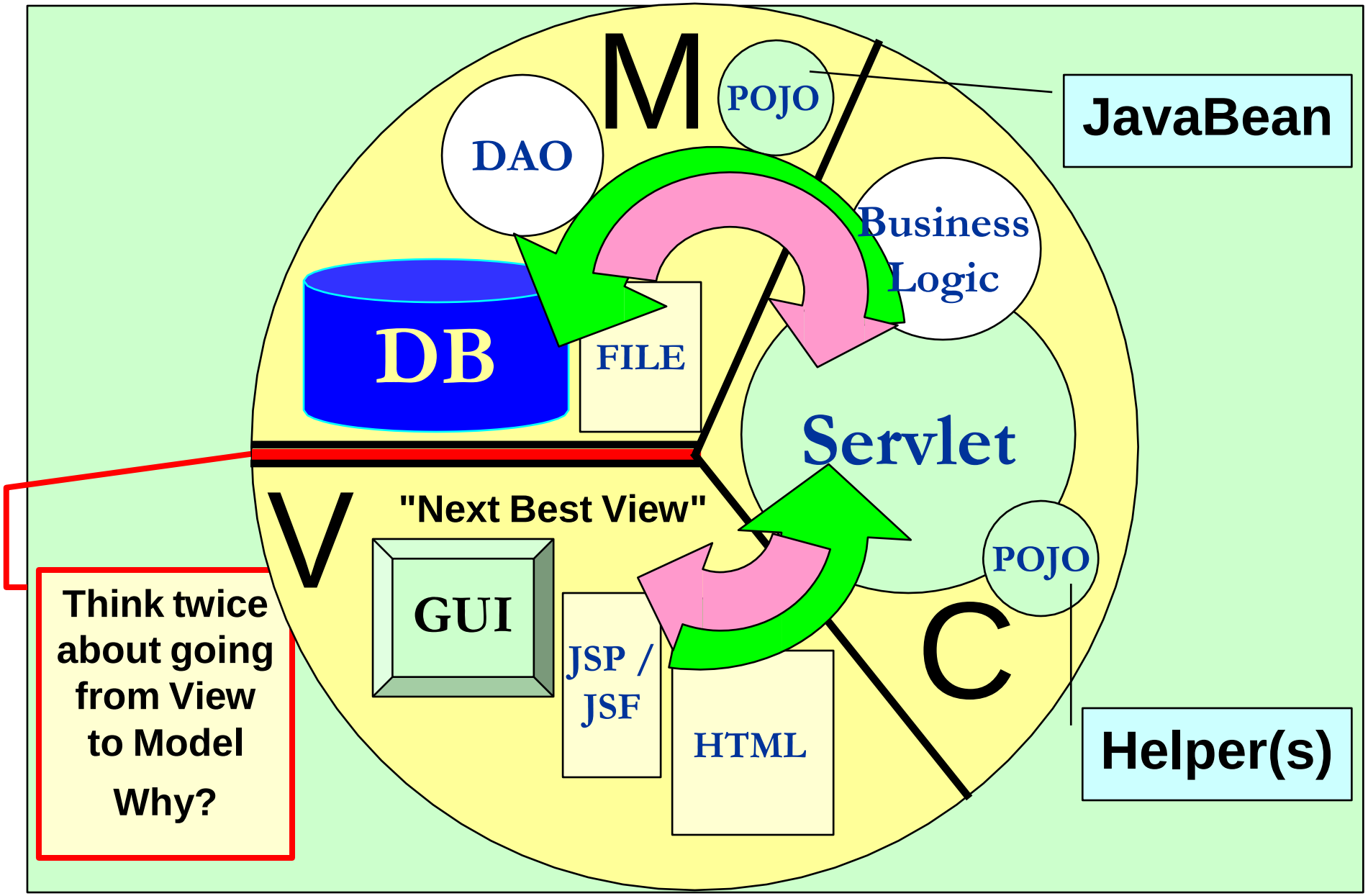
What is Spring MVC?

- A Java Web app framework based on **M**odel **V**iew **C**ontroller (MVC)
  - Originally created due to shortcomings of Servlets/JSP
    - Current JSF (JavaServer faces) has much capability, but some are not happy with it
  - If you're using Spring, Spring MVC is a viable alternative
- Overall goals for Spring MVC
  - **Simplify** web app creation
  - Provide a **strong MVC framework** with clearly defined roles (controller, command object, view resolver, etc)
    - Thus encouraging good architecture
  - **Integrate** easily with the rest of Spring
  - Build on **standard Java EE** (servlets/JSP, custom tags, JavaBeans)
  - Flexibly integrate with additional template technologies (e.g. Velocity)

General MVC Architecture

- **M**odel **V**iew **C**ontroller
  - Standard pattern for creating applications with user interfaces
- **Model**: Representation of the business domain
  - Independent of user interface or application flow
  - In Java EE typically consisting of data, business objects, JavaBeans for storing state and data transfer, and other components (e.g. EJB)
- **View**: Way of representing the data to a human user
  - In Java EE something that generates HTML
    - Often a **JSP** or **JSF** page
    - Other resources such as images, charts, etc.
- **Controller**: Intermediary between Model and View
  - In Java EE **Servlets** are the core controller components
    - May be encapsulated, such as with JSF controllers
  - Controls user flow through the application, and invokes business logic

MVC Pattern flow

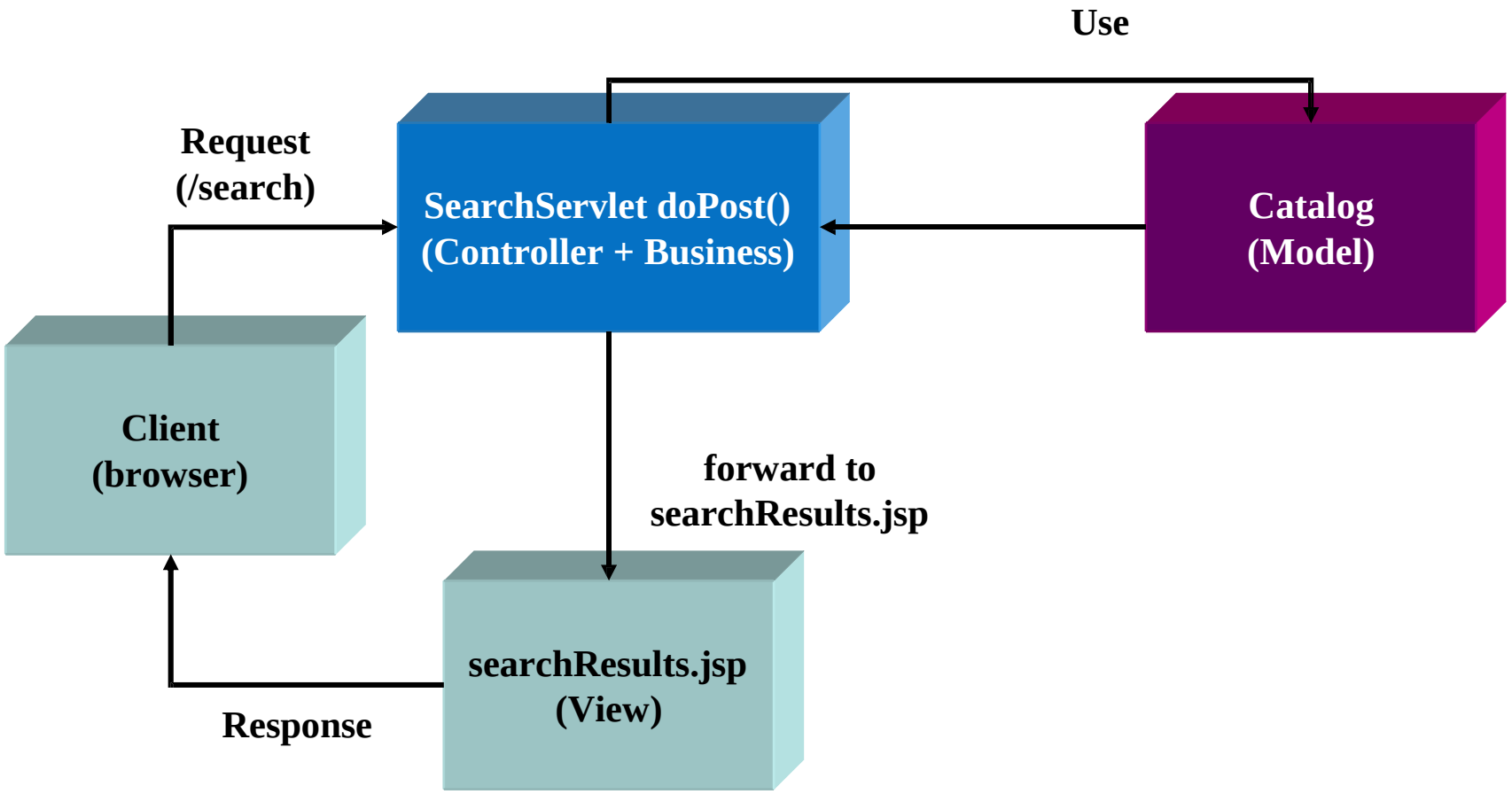


Spring MVC Architecture

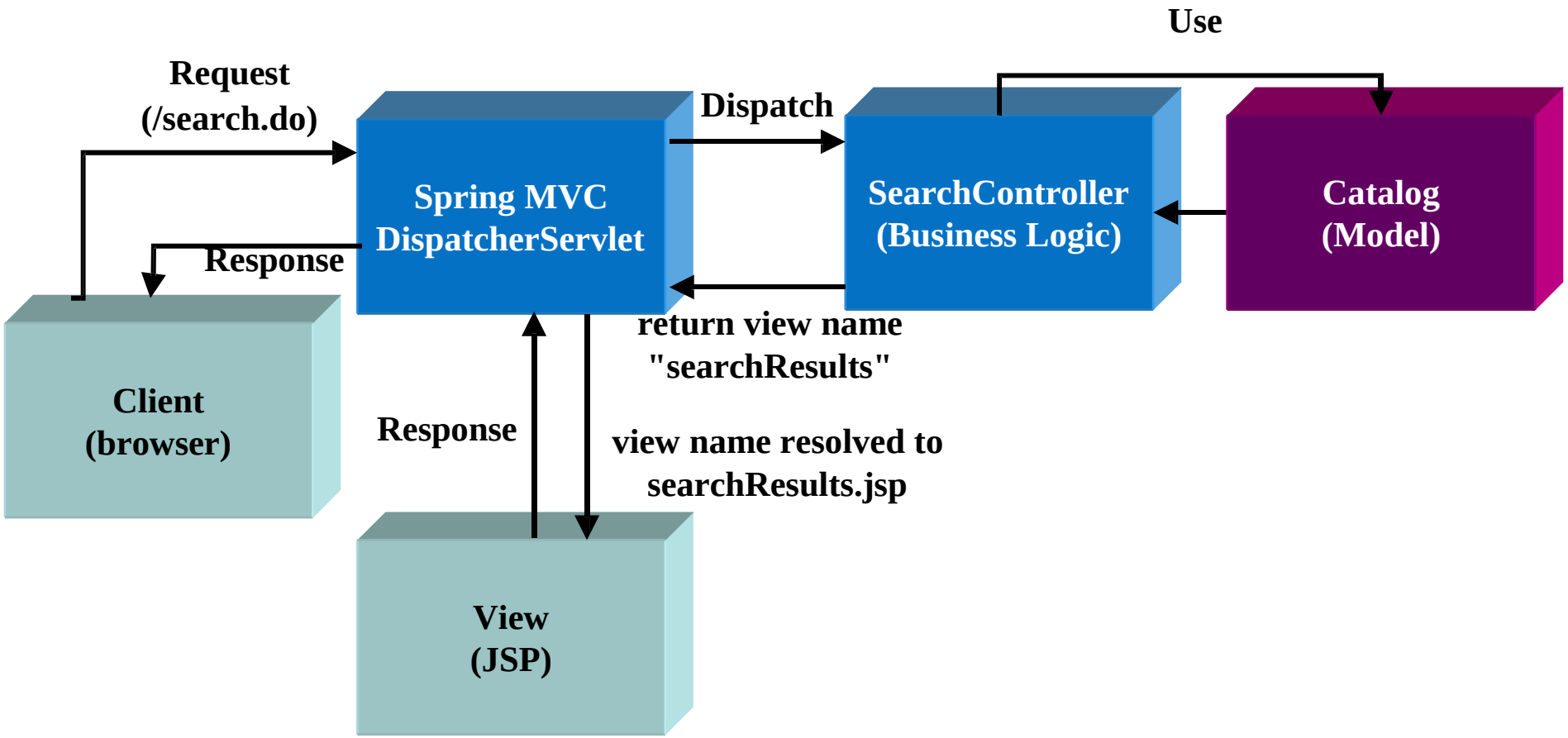
- Spring uses a **front controller** architecture
  - The front controller is a central servlet that receives all requests, and dispatches them to other components (controllers) to handle

- The main components of a Spring MVC application are:
  - ***DispatcherServlet***: The front controller servlet
  - ***Controllers***: Handle specific user requests, generate the data for response, return a response to determine the view
  - **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

Simple Search App Model - Servlets/JSP



Simple Search App Model - Spring MVC



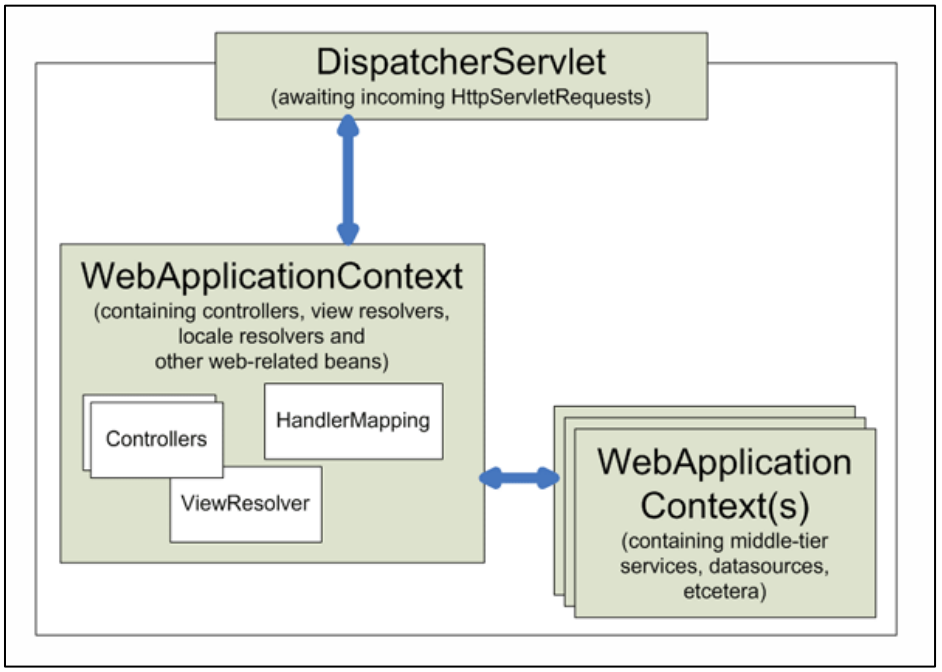
DispatcherServlet

- ***DispatcherServlet***: Front controller for Spring MVC
  - The integration point between Spring MVC and the Web container
  - It's a standard servlet
- *DispatcherServlet* must be configured in *web.xml*
  - To map appropriate requests to the *DispatcherServlet*
  - Typically, a pattern such as ***.do*** is used
  - Requests with a .do suffix (e.g. ***logon.do***) are handled by Spring MVC

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

DispatcherServlet Initialization

- The *DispatcherServlet* has its own *WebApplicationContext*
  - Initialized by the framework when the servlet is initialized
  - The default config file is: WEB-INF***[servlet-name]-servlet.xml***
  - In our example, it will look for WEB-INF***springmvc-servlet.xml***
  - This file **must** be present for Spring MVC to initialize



Controllers

- **Controllers**: Spring MVC component to **handle requests**
  - Dispatcher servlet forwards requests to controllers
  - Shields you from low level APIs (e.g. servlet API), interprets user input, generates a model, and directs response to a view
- Controllers are **Annotation-based** POJOs - the API includes:
  - ***@Controller***: Indicates the annotated class is a web controller
    - Stereotype of *@Component* - usually auto-detected via scanning
    - In package ***org.springframework.stereotype***
  - ***@RequestMapping***: Maps requests to a controller class/method
    - In package ***org.springframework.web.bind.annotation***
  - ***RequestMethod***: enum of HTTP request methods (e.g. GET)
- **Component scanning** must be enabled to use these
  - Using ***context:component-scan*** as seen earlier

A Very Simple Controller

- The simple Logon*Controller* below (handling a logon), receives requests of the form ***/logon.do***
  - ***@RequestMapping("/logon")*** specifies the base URL handled
  - ***@RequestMapping(method = RequestMethod.GET)*** specifies that ***get()*** will be used to handle HTTP GET requests
  - The return value ***/jsp/logon.jsp*** specifies the new view for the user

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/logon")
public class LogonController {

  @RequestMapping(method = RequestMethod.GET)
  public String get() {
    return "/jsp/logon.jsp";
  }
}
```

Control Flow

- A */logon.do* request results in the following flow
  - ***DispatcherServlet* receives** the request
    - Based on the *.do mapping in *web.xml*
  - It forwards the request to ***LogonController.get()***
    - Based on the *@Controller* and *@RequestMapping* in the controller
    - *get()* returns a value of */jsp/logon.jsp*
  - *DispatcherServlet* generates the resulting view
    - Based on the value */jsp/logon.jsp*
    - Generally, logical view names are used, not file names or paths
    - We'll see this being done shortly
- That's all there is for this simple controller
  - We'll look at more functionality soon

LEARNINGPATTERNS

Lab 6.2 – Spring MVC Basics

Lab 6.2 – Spring MVC Basics

Lab

- **Overview**: In this lab, we'll use some very basic Spring MVC capabilities to implement the same functionality as the last lab
  - We'll first set up the *DispatcherServlet* to handle web requests
  - Next we'll create a very simple controller to handle a request for a search form
  - We'll run our application to see these pieces at work, and in future labs we'll enhance it to use more Spring MVC capability
- **Objectives**:
  - Configure *DispatcherServlet*
  - Create a controller
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab06.2

Tasks to Perform

- Remove Lab06.1 from the server *, and close all open files/projects
- Create a new Dynamic Web project called Lab06.2
  - See Lab 6.1 instructions if you need more detail on this
  - Remember to **set the context root** to javatunes
- Right click on the project in Project Explorer and select **Build Path | Configure Build Path** and add in the Spring library as before
  - Next, click **Deployment Assembly** in the left pane (for configuring Web app deployment dependencies)
  - Click **Apply** if prompted to save modifications
  - Click **Add | Java Build Path Entries** click **Next**, select the **Spring** library, click **Finish**, then **OK**

Configure DispatcherServlet

Lab

Tasks to Perform

- Open *WEB-INF**web.xml** for review and editing
  - See notes on logging configuration review
  - Locate the `<servlet>` element for the springmvc servlet and add in the following value for the `<servlet-class>`

```
org.springframework.web.servlet.DispatcherServlet
```

  - Locate the `<servlet-mapping>` element for the javatunes servlet and add in the following value for the `<url-pattern>`

```
*.do
```

  - This maps the dispatcher servlet to any URL ending with *.do
- Open *WEB-INF**springmvc-servlet.xml** for editing
  - Note how Spring's context namespace for is already included
  - Add in the XML element for the `component-scan` elements and specify the scanning of the `com.javatunes` packages (see notes)

Configure DispatcherServlet

Lab

Tasks to Perform

- **Open** *com.javatunes.web.SearchController* for editing
  - It will handle HTTP GET requests to */search.do* URLs
- Add annotations to *SearchController* to declare it a controller class, and to map it to the request */search.do*
  - You will need two annotations
- Add annotations to *get()* so that it handles an HTTP GET request
  - Have it return a value so the */jsp/searchForm.jsp* page is displayed as the view
  - This JSP page is already given to you, and includes a search form for performing a search, which we'll use in the next lab
- To deploy to the server, right click on the Lab06.2 project, and select **Run As | Run on Server** and deploy as in Lab 6.1

Test Your Results

Lab

Tasks to Perform

- A browser should open on: *http://localhost:<8080>/javatunes*
  - This shows a very simple home page that contains a link to /search.do
  - Click this link, it should take you to the search form JSP page
  - This shows that your controller class is being used
- When the link for search.do is clicked, the dispatcher servlet receives the request (a GET), and delegates it to your *SearchController.get()* method
  - The return value of *get()* is used to generate the view
- You can see how simple it is to build a controller using Spring MVC
- Note: **Doing a search is NOT yet implemented**
  - It will not work until the next lab

STOP

@RequestParam – Parameter Binding

- **@RequestParam** binds request parameters to controller method arguments
- Assume two request parameters for a logon: name and password
  - They can be bound to method args as shown in the example below
- ***@RequestParam*** has two other optional elements (see notes)
  - ***required***: Whether parameter is required
  - ***defaultValue***: Default value if no value on request

Example: ***@RequestParam(value="password", required="false")***

```
// Part of LogonController - most detail omitted
@RequestMapping(method = RequestMethod.POST)
public String processLogon(
   @RequestParam("name") String name,
   @RequestParam("password") String password) {
   // Do logon processing - all detail omitted
   return "/jsp/welcome.jsp";
}
```

Returning Model Data

- Handler methods can return model data for use by the view
  - A number of model classes are supplied to hold this data
- *ModelAndView* holds both view information and model data
  - It's in package (*org.springframework.web.servlet*
  - Generally used as a return value from a controller method
  - Has constructors/methods to get/set view, & add attributes to the model
  - Some common constructors / methods include:
  - *ModelAndView(String viewName, String modelName, Object modelObject)*
    - Create an instance with the given view name, model name, and model data
  - *ModelAndView addObject(String attributeName, Object attributeValue)*
    - Add an attribute to the model with the given name
  - See notes for more methods

Returning Model Data

- The example below uses the constructor shown previously to create a *ModelAndView* instance and returns it
- *DispatcherServlet* will do the following
  - Extract the data from the model map
  - Make model attributes available to the view
    - e.g. put them on the request where they can be accessed via JSP EL variables
  - Forward to a view page based on the view value

```
@RequestMapping(method = RequestMethod.POST)
public ModelAndView processLogon(
   @RequestParam("name") String name,
   @RequestParam("password") String password) {
   // Assume checkLogon defined elsewhere - detail not shown
   Principal p = checkLogon(name,password);
   return new ModelAndView("/jsp/welcome.jsp", "principal", p);
}
```

The JSP Pages

- Below, we show a logon form and the resulting welcome page
  - At this point, they're plain JSP pages, with no Spring functionality
- The logon submission form:
  - Submits to **logon.do**
  - Includes **name** and **password** parameters
- The welcome page uses the Principal object that the controller method adds to the model

```
<!-- Logon submission page -->
<form method='post'
        action='${pageContext.request.contextPath}/logon.do'>
  <input size='20' type='text' name='name'/>
  <input size='20' type='password' name='password'/>
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

```
<!-- Welcome page -->
Welcome ${principal.name}
```

Other Handler Method Capabilities

- Handler methods annotated with *@RequestMapping* allow other arguments that will be automatically initialized, including:
  - **Request** or **response** objects from Servlet API
  - **Module** object: The *Http Module*
  - *java.util.Locale*: Current request Locale
  - *InputStream*/*Reader*: Raw stream/reader exposed by servlets
  - *OutputStream*/*Writer*: Raw stream/writer exposed by servlets
  - *@RequestHeader* annotated parameter: Specific header
  - *@CookieValue* annotated parameter: Specific cookie
- Return values can include:
  - *ModelAndView*, as seen earlier
  - *Model* object, with view name implicitly generated from requests
  - *Map* object, with view name implicitly generated from request

Other Handler Method Capabilities

– ***View*** object, with model implicitly determined

Example – Other Handler Capabilities

- In the example below, we show an example of using some of the additional capabilities of handler methods

```
// Access HttpServletRequest and HttpServletResonse Objects
@RequestMapping("/getServletObjects")
public void displayServletInfo(
   HttpServletRequest request, HttpServletResponse response)
   { System.out.println(request.getParameter("password"); }


// Access Module
object and cookie with Module
id
@RequestMapping("/display Module
Info") public void display Module
Info(
   Http Module
   the Module
   , @CookieValue("J MODULE
   ID") String sessID)
   { /* … */ }
```

Example – Other Handler Capabilities

```
// Access request header values
@RequestMapping("/displayHeaderInfo")
public void displayHeaderInfo(
    @RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Keep-Alive") long keepAlive) { /* … */ }
```

Lab 6.3 – Parameters and Models

Lab 6.3 – Parameters and Models

Lab

- **Overview**: In this lab, we'll extend the functionality of our web app to do the following:
  - Extract a request parameter
  - Return a model object that will be used to generate the resulting view
- **Objectives**:
  - Extract a request parameter with *@RequestParam*
  - Use *ModelAndView* to return a model
- **Builds on previous labs**: Lab 6.2
  - Continue working in your Lab06.2 project
- **Approximate Time**: 20-30 minutes

Finish the Controller

Lab

Tasks to Perform

- **Open** *com.javatunes.web.SearchController* for editing
- We configure a Catalog bean, as seen in earlier labs, in the Spring context
  - Inject this bean into *SearchController* using *@Autowired*
    - Just declare a field of type Catalog, and apply the *Autowired* annotation to it * – we'll use it in our handler method
- Add a new method, *processSearch()* to the controller
  - It should be called for an HTTP POST
  - It should have a *String* argument to hold the keyword, that is initialized by a "keyword" parameter on the request
  - Generate the search results by calling *Catalog.findByKeyword* on the catalog object using the keyword parameter
  - If should return a *ModelAndView* object that
    - Specifies a view of *jsp/searchResults.jsp*
    - Includes a model attribute with the name *results* and the search results

Finish JSP and Run

Lab

Tasks to Perform

- Open *jsp/searchHeader.jspf*, and set the action to be ***search.do*** (which will cause the form submission to end up at our SearchController to handle)
- **Restart** the server so your changes are picked up
  - Right click on the server, select **Restart**
- Browse back to: *http://localhost:<8080>/javatunes*
  - Click on the link for the search form, then type in the letter "a" to search on and submit the form
  - The DispatcherServlet will invoke *SearchController*, which will use a catalog do the search, and then return a *ModelAndView* instance initialized with the collection (with name "results") and a view
  - The *DispatcherSerlvet* will then use the *ModelAndView* instance to put the collection on the request with the name "results", and forward to the view
- You should see your search results displayed

STOP

View Resolvers

- It's better practice to use **logical view names** (e.g. *welcome*), rather than hard-coded pages (e.g. */jsp/welcome.jsp*)
- **View Resolvers** in Spring MVC support this
  - Used by the dispatcher to map logical names to actual files
  - ***InternalResourceViewResolver*** is a commonly used one
    - In package: *org.springframework.web.servlet.view*
  - Lets you specify a standard **prefix** / **suffix** added to the logical names to determine the final view page
  - In the example below, this resolver maps ***searchResults*** to the view page ***/jsp/searchResults.jsp***

```
<bean id="jspViewResolver" class=
    "org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

Controller Using Logical Names

- Controllers now use logical view names (not file names)
  - Below, the two controller methods return logical view names
  - These will be mapped by the view resolver to actual files
  - Note that *get()* is used to initially generate the logon view

```
@Controller
@RequestMapping("/logon")
public class LogonController {

  @RequestMapping(method = RequestMethod.GET)
  public String get() { return "logon"; }

  @RequestMapping(method = RequestMethod.POST)
  public ModelAndView processLogon(
    @RequestParam("name") String name,
    @RequestParam("password") String password) {
    // Assume checkLogon defined elsewhere - detail not shown
    Principal p = checkLogon(name,password);
    return new ModelAndView("welcome", "principal", p);
  }
}
```

Other View Resolvers

- ***BeanNameViewResolver***: Maps a view name to a bean
  - For rendering view with a Java class, rather than JSP - see example at bottom which maps view name "*welcome*" to the *WelcomeView* bean
- ***XmlViewResolver***: Uses external XML file to map views to beans
  - Externalizes bean mappings for views to an XML file
  - Default XML file is */WEB-INF/views.xml*
- ***ResourceBundleViewResolver***: Uses external properties files to map views to beans
  - Allows for localization / internationalization

```
<!-- Configure the bean name view resolver -->
<bean  id="viewResolver"
  class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

<!-- WelcomeView class used to generate view for view name "welcome" -->
<bean  id="welcome"  class="com.javatunes.web.WelcomeView"/>
```

LEARNINGPATTERNS

Lab 6.4 – View Resolvers

Lab 6.4 – View Resolvers

Lab

- **Overview**: In this lab, we'll add a view resolver to our configuration
  - The controller will now use logical view names, rather than actual JSP pages
- **Objectives**:
  - Work with *InternalResourceViewResolver*
  - Use logical names in handler methods
- **Builds on previous labs**: Lab 6.3
  - Continue working in your Lab06.2 project
- **Approximate Time**: 15-20 minutes

Add a View Resolver

Lab

Tasks to Perform

- Open *WEB-INF\springmvc-servlet.xml* for editing
  - Add a definition for an *InternalResourceViewResolver* bean
  - Use a prefix of */jsp/*, and a suffix of *.jsp*
  - Maps, for example, the logical view name *search* to */jsp/search.jsp*
- **Open** *com.javatunes.web.SearchController* for editing
  - Modify both the *get()* and *processSearch()* methods to return logical view names instead of hard coded JPS pages
- Restart the server (Right click on server, Restart)
- Test your Web app again starting from *http://localhost:8080/javatunes*
  - It should all work exactly as before

STOP

LEARNINGPATTERNS

Forms and Command Objects

Integration with Java EE

Spring MVC Basics

Forms and Command Objects

Spring MVC Form Tags

- Spring MVC uses form tags to give forms access to model attributes
  - *<form:form>* renders an HTML 'form' tag and exposes a model object as a binding path to inner tags for binding
    - *modelAttribute='logon'* exposes the logon object within the form *
    - It puts the object in the *PageContext* for access by inner tags
  - *<form:input>* renders an HTML 'input' text tag
    - *path='name'* binds that input element to the *logon.name* property
  - The initial form view **must** be generated via a controller, and the form's action comes from the URL associated with that controller
    - *LogonController* uses *get()* to direct a GET of *logon.do* to this page

```
<!-- Logon submission page /jsp/logon.jsp -->
<%@ taglib prefix="form"
        uri="http://www.springframework.org/tags/form"  %>
<form:form modelAttribute='logon'>
  <form:input size='20' path='name'/>
  <form:input size='20' path='password'/>
  <input type='submit' name='Submit' value='Logon'/>
</form>
```

A JavaBean Command Class

- Spring MVC "command classes" hold form data and possibly model data *
  - The *Logon* class below holds a logon name, password, and a *Principal* object (the result of a successful logon)
  - It is a POJO that follows JavaBean conventions for properties
    - With three properties and their get/set methods

```
package com.javatunes.web;
import    java.security.Principal;

public class Logon {
   private String name;
   private String password;
   private Principal principal;

   public String getName() {return name;}
   public void setName(String nameIn) {this.name = nameIn;}
   // Other get/set methods not shown
}
```

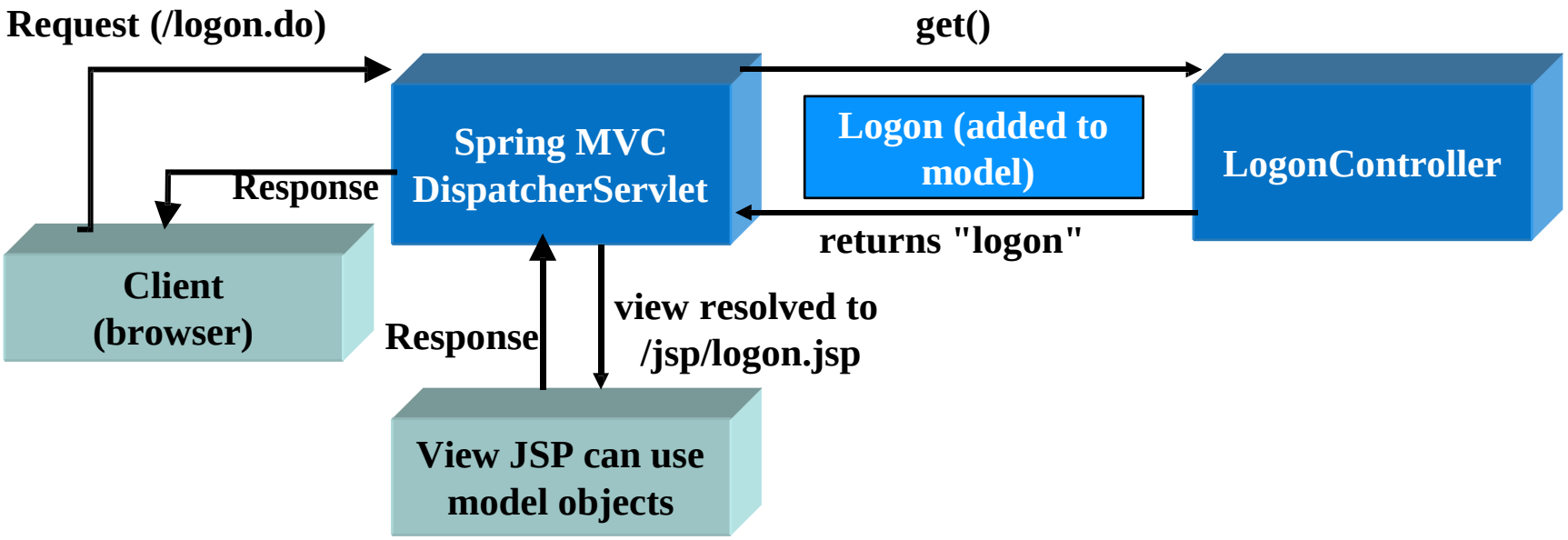
Working with Model Classes

- *@ModelAttribute* binds a method parameter to a model attribute
  - Used to to expose a command object to a web view by annotating handler method parameters
  - This will cause a command object to be created (if not already present in the model) and bound to the parameter
- The example below binds an instance of *Logon* into the model
  - It is available to the Web views – note how we initialize it in *get()*

```
@RequestMapping(method = RequestMethod.GET)
public String get(@ModelAttribute("logon") Logon l) {
   l.setName("Jane Doe");  // Can initialize the model if you want
   return "logon";    // Return the view name
}

@RequestMapping(method = RequestMethod.POST)
public String processLogon(@ModelAttribute("logon") Logon l) {
   // Assume checkLogon defined elsewhere - detail not shown
   l.setPrincipal(   checkLogon(l.getName(),l.getPassword())   );
   return "welcome";
}
```

Request Handling Flow

- The diagram below shows the flow for initial form rendering
- The JSP at bottom shows *welcome.jsp* (result of a logon)
  - Note: The principal object is now a property in the *logon* bean
  - Before the principal basically was the model




```
<!-- Welcome page - previously used ${principal.name} -->
Welcome  ${logon.principal.name}
```

Exposing Reference Data in the Model

- *@ModelAttribute* can be used to expose reference data
  - By annotating a controller method, as shown below
  - The example adds a model attribute "*previousNames*"
    - Containing a list of previous logon names - used to populate a select
  - The *Logon* class would need a *previousName* property for this

```
@ModelAttribute("previousNames")
public Collection<String> populateLogonNames() {
   ArrayList<String> names = new ArrayList<String>();
   names.add("");
   names.add("James  Gosling");
   names.add("Jimmy G");
   return names;
}
```

```
<form:form modelAttribute='logon'>
   <form:input size='20' path='name'/>
   <form:input size='20' path='password'/>
   <form:select path="previousName" items="${previousNames}"/>
   <input type='submit' name='Submit' value='Logon'/>
</form>
```

LEARNINGPATTERNS

Lab 6.5 – Forms and Command Objects

Lab 6.5 – Forms and Command Objects

Lab

- **Overview**: In this lab, we'll expand on the Spring MVC functionality that we use by using the form tag library, and an associated command object to handle the form submission
  - We'll modify our controller to use the command objects
  - This will be used in our JavaTunes web app to handle the search functionality
- **Objectives**:
  - Use the Spring MVC form tags
  - Bind model attributes (command objects) to request parameters
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab06.5

Tasks to Perform

- Remove Lab06.2 from the server *, and close all open files/projects
- Create a new Dynamic Web project called Lab06.5
  - See Lab 6.1 instructions if you need more detail on this
  - Remember to **set the context root** to javatunes
- Right click on the project in Project Explorer and select **Build Path | Configure Build Path** and add in the Spring library as before
  - Next, click **Deployment Assembly** in the left pane (for configuring Web app deployment dependencies)
  - Click **Apply** if prompted to save modifications
  - Click **Add | Java Build Path Entries** click **Next**, select the **Spring** library, click **Finish**, then **OK**

Finish the Controller

Lab

Tasks to Perform

- Open *com.javatunes.web.**Search*** for review
  - This is a command class with three properties – a keyword (*String*), artist (*String*) and search results (*Collection<MusicItem>*)
- Open *com.javatunes.web.**SearchController*** for editing
  - Note how *get()* and *processSearch()* both have parameters of type Search
  - Annotate the parameters in both methods with *ModelAttribute* so that they are bound to a model attribute named "search"
- Finish *get()* by initializing the keyword to "*Diva*"
  - This value will show up in the initial rendering of the form
- Finish *processSearch()* as follows:
  - Extract the keyword from the search object, and use it to search
  - Add the results into the search object

Finish the JSP Pages

Lab

Tasks to Perform

- Open *jsp\searchHeader.jspf* for editing (it contains the search form)
  - Note the taglib for the Spring MVC form tag library at the top
  - Modify the opening *form* tag to include the form tag library prefix, and to bind to the command with a name of "*search*"
  - Modify the closing *form* tag to include the form tag library prefix
  - Finish the *form:input* tag's path attribute to bind to the "keyword" property of the command object

- Open *jsp\searchResults.jsp* for editing (the search results)
  - The search command bean is available in the results view also
  - Finish the *c:forEach* tag's items attribute by initializing it to refer to the results property of the search bean

Form Initialized and Rendered

Lab

- We can see the results below
  - We've browsed to *search.do*, and the response to that request has been generated by the form in *searchForm.jsp*
  - Note how the keyword field has been initialized to "Diva"

Test Your Application

Lab

Tasks to Perform

- Deploy to the server as you did earlier (restart the server)
- Open a browser on: *http://localhost:<8080>/javatunes*
  - Click the link to go to the search form – you should see your default search term of "Diva"
  - Do a search – you should see the same results we've seen before
  - Congratulations - you've made a simple, but complete, Spring MVC app
- **Optional**: Add reference model data to the application
  - In *SearchController*, annotate *populateArtists()* so it adds an attribute named "*artists*" into the model
  - In *SearchController*, add code to *processSearch()* to test for and use the artist property if it has a value of greater than 0 length
  - In searchHeader.jspf, uncomment/complete the *form:select* to bind to the artist property, and initialize the items from the artists attribute
  - Run the application again and test this functionality

STOP

Review Questions

- How do you integrate the Spring container with normal web applications?
- What is MVC?
- What are the architecture and main components of Spring MVC?
- How do you configure Spring's *DispatcherServlet*?
  - What does its URL mapping look like?
- What does a command controller do?
  - How is it invoked?
- How do controllers help you process forms?
  - What role to Spring command classes play in this?

Lesson Summary

- The Spring container integrates with normal Web applications via Springs ***ContextLoaderListener***
  - It automatically loads Spring's root application context and makes it available to Web apps
- **MVC**: A common pattern for organizing apps with user interfaces into three areas of functionality
  - **Model**: Representing the business domain
  - **View**: Presenting the data to users
  - **Controller**: An intermediary between the model and the view
- Spring implements MVC as follows:
  - Spring beans used as model
  - Normal view pages (e.g. JSP) as view
  - Command controllers as its controllers

Lesson Summary

- Spring's ***DispatcherServlet*** is a normal servlet configured in *web.xml*
  - Its url-mapping uses patterns, e.g. *.do
  - This mapping is applicable to a whole category of requests
  - It acts as a front controller to the web app
- **Controllers** handle incoming requests
  - They access request data, execute business logic, set up the model with data, and indicate the next view to render
- Controllers can easily process form requests
  - They can bind method parameters to command classes to gain access to the form data
- **Annotations** are considered best practice for configuration
  - They make configuration very simple

LEARNINGPATTERNS

Module 7: Spring Security Overview

Spring Security

- Provides security services for Spring / Java EE apps
- Comprehensive support for **authentication** (identifying users)
  - HTTP Basic, Digest, Form-Based, X.509 client certificate
  - Integration with common containers (Tomcat, JBoss, etc)
  - LDAP, JAAS, and much more
- Authentication supported in different areas
  - Authorizing Web requests
  - Authorizing invocations on a bean
  - Accessing domain objects
- Fairly simple XML configuration
  - Much improved from previous releases

Spring Web Security - web.xml

- Spring security can be easily used to protect Web apps
  - Via the *web.xml* and Spring configuration files
- ***web.xml*** uses a filter specifying the URLS to protect
  - This provides a hook into the Spring security system
  - Below, we use ***DelegatingFilterProxy*** (a supplied Spring class) to protect all URLS in the Web app
    - Via a **/*** url-pattern

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Spring Web Security - Spring Configuration

- Web security is configured in the Spring config file
  - Using the ***<http>*** element (in the Spring security namespace)
  - Below, ***<http>*** is used with the ***auto-config*** attribute
    - This sets up a default configuration to enable security easily
  - ***<intercept-url>*** specifies request URLS to apply security to
    - ***/**** specifies all URLs in the application
    - The ***access*** attribute requires users to be in the ROLE_USER role

```
<?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:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/context/spring-security-3.0.xsd">

  <security:http auto-config="true">
    <security:intercept-url pattern="/**" access="ROLE_USER" />
  </security:http>
</beans>
```

Spring Web Security - Spring Configuration

- You define users with an **authentication provider**
  - These check the validity of credentials (e.g. name/password)
  - Configured in XML via ***<authentication-provider>***
- Authentication providers depend on a **user details service** to get/store authentication information
  - The ***user-service*** variant defines this info in the config file *
  - Below, we define users jane and john with passwords and roles

```
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="jane" password="password"
                     authorities="ROLE_USER, ROLE_ADMIN" />
      <security:user name="john" password="password"
                     authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

More <security:http> Capabilities

- *<http>* has a number of configuration attributes, including
  - ***lowercase-comparisons*** : Convert URLs to lower case?
  - ***realm***: Realm name if basic authentication is used
  - ***access-denied-page***: Forwarding page for failed access
- *<http>* also has sub-elements for configuration, including:
  - ***<http-basic>***: Basic authentication (used if forms not enabled)
  - ***<form-login>***: Form login configuration, including login page, login processing url, default target, url, etc.
  - ***<anonymous>***: Enables anonymous principals
  - ***<logout>***: Enables logout and its configuration
  - ***<intercept-url>***: URL patterns for security - with attributes:
    - ***pattern***: The URL path to match
    - ***method***: HTTP method to match (used in combination with pattern)
    - ***access***: Roles to allow access to
    - ***channel***: Can be "http", "https", or "any" to define access allowed

Other User Details Services

- **Database-based** variants are available
  - User/authority info stored in a DB with a specified schema *
  - Provided by jdbc-user-service
  - Easily accessed by configuring a datasource, then using it in the authentication provide

```
<!-- securityDataSource configured elsewhere -->
<security:authentication-provider>
  <security:jdbc-user-service
            data-source-ref="securityDataSource"/>
</security:authentication-provider>
```

- **LDAP-based** variants are available
  - Can configure mapping of user name to LDAP DN
  - Can configure binding of user to LDAP directory
  - Can configure mapping of LDAP groups to authorities

LEARNINGPATTERNS

Lab 7.1 – Spring Web Security

Lab 7.1 – Spring Web Security

Lab

- **Overview**: In this lab, we'll explore Spring security
  - We'll use it to add security to the JavaTunes Web application form previous labs
- **Objectives**:
  - Become more familiar with Spring security
  - Use Spring security to enforce access to a Web application
- **Builds on previous labs**: none
- **Approximate Time**: 20-30 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab07.1

Tasks to Perform

- Remove Lab06.5 from the server *, and close all open files/projects
- Create a new Dynamic Web project called Lab07.1
  - Remember to set the context root to javatunes
- Right click on the project in Project Explorer and select **Build Path | Configure Build Path** and **add in the Spring library** as before
  - Also add all the jars in *StudentWork\Spring\SpringSecurityLibraries*
    - See notes for details on adding the jars
    - These jars are a subset of the full Spring security distribution
  - Also add the **Spring library and all the spring security jars** via the **Deployment Assembly** (see notes for details)

Configure Application

Lab

Tasks to Perform

- Open *WebContent\WEB-INF**web.xml*** for editing
  - Find the *<filter>* element for the Spring security filter
  - Finish *<filter-mapping>* so security is applied to all requests
- Open *WebContent\WEB-INF**applicationContext.xml*** for editing
  - Ignore errors in this file until you finish this part of the lab
  - Note the security namespace declaration at the top
  - Finish ***<security:http>*** so auto-config is true
  - Finish ***<security:intercept-url>*** so it applies to all requests, and access is granted to users in ROLE_USER
  - Find the ***<security:authentication-provider>***
  - Finish the two user elements with name, password, & authorities
  - At least one user should contain the ROLE_USER authority

Test Your Application

Lab

Tasks to Perform

- Deploy to the server as you did in Lab 6.1 (restart the server)
- Browse to: *http://localhost:<8080>/javatunes*
  - You'll be prompted for a username/password - enter the data for the user that has the ROLE_USER authority
  - You should be forwarded to the JavaTunes home page
- Try logging out (browse to **javatunes/j_spring_security_logout**)
  - Log in again with invalid information, or with a non ROLE_USER user
  - You should be prompted for another login
- Add ROLE_ANONYMOUS to the roles in *applicationContext.xml*
  - Redeploy, and access the app in the browser - you should be able to access it without logging in now (you should log out first)
- **Optional**: set ***auto-config="false"*** in *<security:http>*
  - Try adding in ***<security:http-basic/>*** to the *<http>* and run the app - you'll see that the login uses HTTP basic now

STOP

© 2007-13 LearningPatterns Inc.
All rights reserved.

Test Your Application

Lab

Method Security

- Method security is very simple, and can be added three ways
  - Via **annotations** on your bean class
  - Via **AspectJ** pointcut expressions
  - Via **XML** configuration on a bean definition

- **Annotations** are enabled via the configuration element below
  - Enables Spring specific annotations (***@Secured***)
  - Enables JSR-250 annotations (***@RolesAllowed***, ***@DenyAll***, ***@PermitAll***)
  - These annotations are usable on your Spring bean classes
  - Access specified in terms of roles in your authentication provider

```
<security:global-method-security secured-annotations="enabled"
                                 jsr250-annotations="enabled"/>
```

Method Security - Annotations

- Below, ***@Secured*** secures a bean class from our labs
  - It specifies that only users in ROLE_USER can access any methods of the class
  - You can also annotate individual methods
  - The security runtime will now check and enforce these access requirements on bean invocations

```
import    org.springframework.security.annotation.Secured;
// Other imports, details omitted
@Secured("ROLE_USER")
public class JavaTunesCatalog implements Catalog { /* … */ }
```

- Below, we show usage of **JSR-250** annotations
  - Useful if you don't want Spring specific code in your beans

```
import    javax.annotation.security.RolesAllowed;
// Other imports, details omitted
@RolesAllowed("ROLE_USER")
public class JavaTunesCatalog implements Catalog { /* … */ }
```

Method Security - Pointcut Expressions

- Pointcut expressions can apply security to a wide-range of code
  - Use *<protect-pointcut>* and the AspectJ expression language in the Spring configuration
  - Below, we declare that only ROLE_USER users can access **any method** in *com.javatunes.catalog.Catalog*
  - Multiple *<protect-pointcut>* elements are allowed
  - That's all that's needed to protect these method invocations
  - This is very powerful, and should be used very carefully

```
<security:global-method-security>
  <security:protect-pointcut
    expression="execution(*  com.javatunes.catalog.Catalog.*(..))"
    access="ROLE_USER"/>
</security:global-method-security>
```

Method Security - XML Configuration

- XML can configure security for a particular bean declaration via **<intercept-methods>**
  - Letting you set protections on an individual bean
  - Below, we specify that all methods in the javaTunesCatalog bean that start with find, can only be accessed by ROLE_USER users
  - This allows simple security configuration within the bean definition
  - Not as powerful as pointcut declarations, but easier to use and understand

```
<bean id="javaTunesCatalog"
        class="com.javatunes.catalog.JavaTunesCatalog">
  <security:intercept-methods>
    <security:protect method="find*" access="ROLE_USER" />
  </security:intercept-methods>
</bean>
```

LEARNINGPATTERNS

Lab 7.2 – Method Security

Lab 7.2 – Method Security

Lab

- **Overview**: In this lab, we'll add method security to a bean
- **Objectives**:
  - Add method security to a bean using Spring Security
- **Builds on previous labs**: 7.1
- **Approximate Time**: 15-25 minutes

Tasks to Perform

- Continue to work in the **Lab07.1** directory
- Open ***applicationContext.xml*** for editing
  - Add in a ***<security:global-method-security>*** element that allows both Spring and JSR250 security annotations
- Open ***JavaTunesCatalog.java*** for editing
  - Add an annotation to allow ROLE_USERS to access the entire bean - use a Spring or JSR-250 annotation

Test Method Security

Lab

Tasks to Perform

- **Restart** Tomcat & browse to http://localhost:8080/javatunes again
  - Login as a ROLES_USER, and do a search - it should succeed
- **Change** the security annotation to require **ROLES_ADMIN** users
  - **Restart** Tomcat and logout (javatunes/**j_spring_security_logout**)
  - Login with a non-admin user, then search - **access will be denied** !
- **[Optional]** Comment/remove the *<global-method-security>* in *applicationContext.xml* and the annotations in *JavaTunesCatalog*
  - In *applicationContext.xml*, add a new *<global-method-security>* with a ***protect-pointcut*** element and a pointcut expression
    - Have it specify that only ROLE_ADMIN users can invoke any *com.javatunes.catalog.Catalog* method
    - See earlier examples in the slides
  - **Restart** Tomcat, login as a non-admin user, then do a search
  - You should see **access denied**

Test Method Security

Lab

Tasks to Perform

- **[Optional]** Comment out the pointcut security declaration
  - In *applicationContext.xml*, add an ***<intercept-methods>*** element to the bean def for *javaTunesCatalog* to specify only ROLE_ADMIN access
  - **Restart** Tomcat, login as a ROLE_USER, then do a search
  - You should see **access denied**

- If Spring security is installed, you can look at the documentation
  - It is likely in a folder like *C:\spring-security-3.1.3.RELEASE*
  - The documentation is in the docs folder

STOP

LEARNINGPATTERNS

Module 8: RESTful Services with Spring

REST Overview

REST and Spring MVC

Client-Side Access to RESTful Service

Common REST Patterns

Module 8: RESTful Services with Spring

- Understand what REST and RESTful services are
- Learn how to implement RESTful services using Spring
- Learn how to create RESTful clients with Spring's RestTemplate
- Create RESTful service implementations, and use them from Java clients

© 2007-13 LearningPatterns Inc. All rights reserved.

LEARNINGPATTERNS

REST Overview

REST Overview

REST and Spring MVC

Client-Side Access to RESTful Service

Common REST Patterns

Overview of REST

- REST stands for **Re**presentational **S**tate **T**ransfer
  - A set of principles on **defining and accessing resources** on a network (e.g. the Web)
  - No official REST specification
  - We'll describe some generally accepted principles here
- REST focuses on **resources** identified by **URI**s (Web links)
  - **Data centric**, not API centric
  - Interactions are **stateless** between requests
  - Resources have a **simple, uniform interface**
    - Limited set of well defined operations - **GET**, **POST**, **PUT**, **DELETE**
    - Defined set of content types (e.g. plain text, XML, JSON)
  - Radically different from RPC (e.g. SOAP)

REST Details

- REST services are **stateless**, and systems communicate by **transferring representations of resources**
  - Every request carries the information needed to carry it out
- RESTful resources are **uniquely identified by URI**
  - Consider the URI to be a **noun** (the resource being accessed)
- REST uses only a few methods - based on HTTP:
  - **GET** - A read - retrieves the resources representation
  - **POST** - A create - creates a new resource
  - **PUT** - An update - pastes a changed copy over the original
  - **DELETE** A delete - Removes the data associated with the resource
  - Consider the method to be a **verb** (the action)

Example: Accessing REST Resources

- URIs are used to access REST-based resources
- Consider a REST service to maintain a list of customers
  - To retrieve all customers, you might use the URI below

http://foo.com/crm/rest/customers

  - **http://foo.com/crm**: The Web app base URI
  - **/rest**: URI for all REST services in this app
    - Mapped in *web.xml* (e.g. the Spring MVC dispatcher servlet)
  - **/customers**: The REST "noun" (the resource to access)
  - **GET**: The verb (assuming this is a GET request)
  - We'll abbreviate this in the form ***<webapp>****/rest/customers*
- A representation of all the customers is returned
  - e.g. the id, name, and address of each customer
  - This could be in many forms – text, XML, JSON, etc

What is a REST Service Interface?

- A set of URIs, associated verbs, their effects, and response
- Consider our customer URI and its possible methods / effects

  <webapp>/rest/customers

  – A **GET** request would return a list of all customers
    - In whatever format the implementers defined (e.g. XML)
  – A **POST** request would create a new customer
    - The request would include the customer information in the request body
- In contrast, a SOAP service might define operations like:
  – getUser, addUser, removeUser, updateUser, listUsers

URI Templates for Client Data

- **URI Templates** are URIs with embedded variables
  - Variables are denoted with ***{variableName}***, as shown below

    *<webapp>/rest/customers/**{id}***

  - The above example is a URI Template
  - ***{id}*** is an embedded variable
  - Clients **fill in the variables with data** to produce a complete URI

    *<webapp>/rest/customers/**123***

- They provide a **structural description** of a service interface
  - They describe the form of the URIs defined for the service
  - Can be used by REST processors to process REST requests
    - A request URI is compared to a URI template to determine the resource being addressed, and to extract the variable data
  - See notes for details

REST URI with a URI Template

- Consider the URI Template below, and the methods it supports
  - It includes an embedded variable for the customer id

<webapp>/rest/customers/{id}

  - e.g. *<webapp>/rest/customers/**123***
  - **GET** returns the customer with the id (potentially as XML)
  - **DELETE** deletes the customer with the id
  - **PUT** modifies the customer with the id using the request contents

- The customer id is extracted from the URI
  - Based on the URI Template that describes where the variable is
  - For the URI above, the GET processor would extract the value 123 as the id
    - Perhaps using it to look up a customer and generate a response

Summary of REST Characteristics

- REST uses **simple representations** (e.g. XML, text, JSON) rather than SOAP messages
- REST has a **small number of fixed methods** (GET, etc)
  - It has many different resources (identified by different URIs)
    - e.g., accessing different customers just a GET on a different URI
- The **interaction is very simple**
  - The request looks like **GET /crm/rest/customers/123**
  - The response is a simple document
- REST exchanges can be cached with existing HTTP mechanisms
- References to other resources can be represented as simple hyperlinks

Comparison of REST and SOAP

- REST **is simpler** than SOAP
  - Uses common concepts and existing infrastructure (HTTP/Web)
  - Small learning curve, and closer in design to the Web itself
  - SOAP is much more complex
- REST is **HTTP based**
  - SOAP is transport agnostic – e.g. SOAP over JMS exists
- REST is basically **point-to-point**
  - SOAP allows for distributed communications with multiple intermediary nodes
- REST **has no standards for security, reliable messaging**
  - However, REST can leverage existing HTTP standards
  - SOAP has a lot of additional capabilities available
- **REST is newer than SOAP**
  - SOAP has become an accepted standard for Web services

LEARNINGPATTERNS

REST and Spring MVC

REST Overview

REST and Spring MVC

Client-Side Access to RESTful Service

Common REST Patterns

Spring Support for REST

- REST support builds on top of Spring MVC, including:
  - ***@RequestMapping*** maps RESTful URIs to standard controllers
  - ***@PathVariable*** supports URI templates
  - ***@RequestBody*** binds incoming HTTP data to Java objects
  - ***@ResponseBody*** binds return values to the response body
  - **HTTP Method conversion** supports GET, POST, DELETE, PUT
  - **New views** support XML and JSON response representations
  - We'll go into detail on most of these

- Generally, you would map the Spring MVC Dispatcher servlet to a URL specifically for REST requests
  - e.g. ***/rest/****
  - All such requests would be dispatched via the dispatcher servlet

A Simple RESTful Controller Example

- Below, we illustrate a simple controller to handle RESTful URIs
  - We'll explain how this works next - some detail is omitted now
- *CustomerController* maps to requests of the form *<webapp>/rest/**customers***
- *getAllCustomers()* maps to URIs of the form ***/customers***
- *getCustomer()* maps to URIs of the form ***/customers/123***
  - The 123 value would be bound to the *Long id* parameter

```
@RequestMapping("/customers")
public class CustomerController {

  @RequestMapping // Find/return all customers
  public String getAllCustomers() { … }

  @RequestMapping("/{id}") // Find/return a single customer
  public String getCustomer(@PathVariable Long id) { … }
}
```

Using @RequestMapping and @PathVariable

- ***@RequestMapping*** maps a REST URI to a controller
  - It supports URI Template matching
  - ***@PathVariable*** binds a controller method parameter to a URI Template variable
- Below, */customers/{id}* specifies a RESTful URI
  - It has a URI Template with a variable named ***id***
  - ***@PathVariable*** binds the id method parameter to this variable
    - Based on matching the variable name to the parameter name
  - This matches a URI of the form *<webapp>/customers/123*
    - In this request, *123* will be bound to the *id* method parameter
    - Generally, a controller uses the parameter values in processing a request, then generates an appropriate response (covered soon)

```
@RequestMapping("/{id}")
public String getCustomer(@PathVariable Long id) { }
```

@ResponseBody for Generating a Response

- ***@ResponseBody*** indicates that a return value should be used to directly generate a response
  - Instead of being used as a view name as in Spring MVC, or as a model object of some sort
- A return value is **automatically converted** to a representation appropriate for the client
  - The representation must satisfy the client request's ***Accept* header**
  - e.g. a ***text/xml*** accept header results in **XML** being returned
  - If there's no Accept header, any representation may be used
  - Spring provides **Http message converters** for this conversion
    - It ships with a number of different converters

@ResponseBody Example

- Below, we show ***@ResponseBody*** applied to *getCustomer()*
  - *getCustomer()* first finds a customer
    - Based on the id extracted from the URI template (detail not shown)
  - It returns the string value of the customer (using *toString()*)
  - The string is used directly to create the response body
    - We use a simple string now to get us started
    - We'll show how to use other representations later

```
@RequestMapping("/{id}")
@ResponseBody
public String getCustomer(@PathVariable Long id) {
   Customer result = … // Find a Customer (not shown)
   return    result.toString();
}
```

Additional URI Mapping Capabiliites

- **Multiple URI Templates** in a URI
  - /customers/fname/{fname}/lname/{lname}
  - /customers/{fname}/{lname}
    - Both URI Templates above have two variables
    - The first example includes literal path elements - a common technique to disambiguate URIs (see notes)
- **Specification of variable names** in *@PathVariable*
  - For when method parameter names don't match variable names
  - The matching is based on the URI Template variable name
  - Below, the *fname* variable value is used to initialize *firstName*

```
@RequestMapping("/fname/{fname}/lname/{lname}")
public String getCustomer(
        @PathVariable("fname") String firstName,
        @PathVariable("lname") String lastName)
{ /* Detail omitted */}
```

LEARNINGPATTERNS

Lab 8.1 – A Simple REST Service

Lab 8.1 – A Simple REST Service

Lab

- **Overview**: In this lab, we will create a simple REST service
  - We'll use URL Templates and *@RequestMapping* to map request URIs to controllers
  - We'll use *@ResponseBody* to indicate that the return value should be used directly to generate the response
  - We'll simply invoke the service from a browser to test it
- **Objectives**:
  - Become more familiar with REST
  - Use Spring MVC to create a simple REST service
  - Test the service using a browser
- **Builds on previous labs**: none
- **Approximate Time**: 25-35 minutes

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab08.1
- We will create a REST service to work with our previous JavaTunes types - it will support two URIs of the forms shown below
  - ***/items*** : Get all the items in the JavaTunes catalog
  - ***/items/5*** : Get the item with the id of 5

Tasks to Perform

- Remove any projects from the server *, close all open files/projects
- Create a new Dynamic Web project called Lab08.1
  - See Lab 6.1 instructions if you need more detail on this
  - Remember to set the context root to javatunes
  - Remember to add the Spring user library to the build path and also via the **Deployment Assembly** (see notes)

Finish the Controller

Lab

Tasks to Perform

- In *web.xml*, review the ***/rest/****** mapping for the dispatcher servlet
- Open *ItemsController.java* (in package *com.javatunes.rest*)
  - Review the class briefly
  - It contains an injected Catalog object
  - It declares two methods that will be used as controllers
- Add annotations to accomplish the following
  - Map the entire class to the ***/items*** URI
  - Configure ***getAllItems()*** as a controller, also mapped to ***/items***
  - Configure ***findItemById()*** as a controller, mapped to a URI of the form ***/items/1***, using a URI Template with a name of id
    - Add an annotation to bind the URI Template variable to the "id" parameter that is already present in the method
  - Annotate both methods so that the return value is used to generate the response

Deploy and Test

Lab

Tasks to Perform

- To deploy to the server, right click on your server in Servers view
  - Select **Add and Remove…**
  - In the next dialog, select the Lab08.1 project, click **Add**, then **Finish**
  - Right click on your server in Servers view again, and select **Restart**
- Open a browser on: *http://localhost:8080/javatunes/**rest/items***
  - This should show the string representation of all the items
  - This is being handled by the *getAllItems()* controller
  - Note that we're using a browser as a client for simplicity in this lab
- Browse to: *http://localhost:8080/javatunes/**rest/items/1***
  - This should show the string representation the item with id == 1
  - This is being handled by the *findItem(Long id)* controller
  - The id parameter is being initialized from the URI template in the request
- It's easy to write and invoke REST services, as you can see

STOP

More Complex Response Objects

- Return data is often more complex than a string
  - For example, a Java object of some type
- These are automatically converted to a suitable representation for the client request (e.g. into XML or Json)
  - Conversion based on the **Accept headers** of the client request
  - Spring MVC provides **HTTP message converter** for this
- Below, *getCustomer()* returns a customer object
  - It is converted to an appropriate representation for the request

```
@RequestMapping("/{id}")
@ResponseBody
public Customer getCustomer(@PathVariable Long id) {
   Customer result = … // Find a Customer (not shown)
   return  result;
}
```

HTTP Message Converters

- Spring ships with a number of message converters
  - Used to convert to/from a representation
  - Common converters include:
    - ***StringHttpMessageConverter***: Reads all media types
      - Writes to *text/plain*
    - ***Jaxb2RootElementHttpMessageConverter***: Reads and writes XML (*text/xml* or *application/xml*)
      - Supports all JAXB annotated objects (*@XmlRootElement*)
      - Registered if **JAXB libraries on classpath** (present in Java 6+)
    - ***MappingJacksonHttpMessageConverter***: Reads and writes JSON (*application/json*) to/from Java objects or *HashMaps*
      - Registered if **Jackson JSON library on classpath**
- HTTP Message Converters must be registered
  - Use ***mvc:annotation-driven*** in your configuration (see notes)

Returning an XML Representation

- Requirements for returning an XML representation from a controller include:
  - JAXB libraries must be on the classpath
    - Almost always true today, since they're part of Java 6 +
  - *Jaxb2RootElementHttpMessageConverter* must be registered
    - Generally via *mvc:annotation-driven* in the config file
  - Request must include *text/xml* or *application/xml* in the Accept header
  - The returned object must be a JAXB type
    - e.g., one annotated with *@XmlRootElement*
- Returning Java **collections** in an XML representation is somewhat cumbersome
  - They have to be wrapped in a JAXB-annotated object
  - We'll show an example of this in the lab

Returning a JSON Representation

- **JSON** (JavaScript Object Notation) is a lightweight data interchange format
  - It is based on the JavaScript object/array structure
- Requirements for returning a JSON representation include:
  - Jackson JSON library must be on the classpath
    - Downloadable from *http://wiki.fasterxml.com/JacksonDownload*
  - ***MappingJacksonHttpMessageConverter*** must be registered
    - Via ***mvc:annotation-driven*** as before
  - Request must include *application/json* in the Accept header
- JSON is more commonly requested from a browser script
  - e.g. by JavaScript code running in a browser

Narrowing Controller Mappings

- *@RequestMapping* supports a ***headers*** element to narrow the primary mapping of a controller
  - Consists of sequence of *My-Header=myValue* pairs
  - A request is matched only when each header has the given value
- *getCustomer()* below is only matched if the request accepts either *text/xml*, *application/xml*, or *application/json*
  - Note: we use ***value=*** to specify the URI (see notes)

```
@RequestMapping(value = "/{id}", headers={
    "Accept=text/xml, application/xml, application/json"})
@ResponseBody
public Customer getCustomer(@PathVariable Long id) {
  Customer result = … // Find a Customer (not shown)
  return result;
}
```

LEARNINGPATTERNS

Lab 8.2 – Returning XML and JSON

Lab 8.2 – Returning XML and JSON

Lab

- **Overview**: In this lab, we will work with a REST service returning more complex representations
  - We'll return an XML representation, which will require annotating the return types with JAXB annotations
  - We'll return a JSON representation, which will require adding the Jackson libraries to the classpath
- **Objectives**:
  - Produce XML and JSON representations from a RESTful service
- **Builds on previous labs**: none
- **Approximate Time**: 25-55 minutes (depending on optional parts done)

Lab Preparation

Lab

- The (new) root lab directory where you will do all your work is:
  C:\StudentWork\Spring\workspace\Lab08.2
- We will work with our previous REST service, and add support for XML and JSON representations

Tasks to Perform

- Remove any projects from the server, close all open files/projects
- Create a new Dynamic Web project called Lab08.2
  - See Lab 6.1 instructions if you need more detail on this
  - Remember to set the context root to javatunes
  - Remember to add the Spring user library to the build path and also via the **Deployment Assembly** (see notes if you need details)
- Review the jars in the Spring library (in your Eclipse project)
  - Note the jars that support JSON (see notes)

Finish the Configuration and Controller

Lab

Tasks to Perform

- Open WEB-INF*springmvc-servlet.xml* (the Spring config file)
  - Add in an *<mvc:annotation-driven>* element to register the message converters we'll be using
  - Note: the *mvc:* namespace is already included
- Open *MusicItem.java* (in package *com.javatunes.util*)
  - Add an *@XmlRootElement* annotation (to enable marshaling to XML)
  - From package *javax.xml.bind.annotation*
- Open *ItemsController.java* for editing
  - Change the return value of *findItem()* from *String* to *MusicItem*

    ```
    public MusicItem findItem(...)
    ```

  - Change the return statement to return the object, not a string

    ```
    return cat.findById(id);
    ```

  - Upon a request, the returned object can now be marshaled to XML

Deploy and Test

Lab

Tasks to Perform

- To deploy to the server, right click on your server in Servers view
  - Select **Add and Remove…**
  - In the next dialog, select the Lab08.2 project, click **Add**, then **Finish**
  - Right click on your server in Servers view again, and select **Restart**
- Browse to: *http://localhost:8080/javatunes/**rest/items/1***
  - This should now show the **XML representation** of the result
  - The return value is JAXB annotated, and the request (from the browser) includes Accept headers for XML (see notes)
- Browse to: *http://localhost:8080/javatunes/**rest/items***
  - This will show the **JSON representation** of the result
  - The return value, a collection, is NOT JAXB annotated, so the JSON conversion is used instead
  - Note the JSON structure

[Optional] Additional Things to try

Lab

Tasks to Perform

- Only do any optional parts if time permits (there are quite a few)
- Open *ObjectWithList.java* for review (in *com.javatunes.util*)
  - *ObjectWithList* is a JAXB annotated wrapper we supply - see notes
- Change the value returned in *getAllItems()*
  - Wrap the collection in an *ObjectWithList* before returning it

```
return  new ObjectWithList(results);
```

- **Restart** the server, and try to browse to *javatunes/rest/items* again
  - You should now see an XML representation of the data (see notes)
- We supply a rest-client program to invoke RESTful services *
  - It allows us more control over the request than a browser
  - You can try it as described below and on the following slides
- Open a command prompt in StudentWork\Spring\rest-client
  - Run the program in the jar file there, e.g.

```
java -jar  restclient-ui-3.1-jar-with-dependencies.jar
```

[Optional] Additional Things to try

Lab

Tasks to Perform

- In the rest-client GUI, do the following
  - Set the URL to *http://localhost:8080/javatunes/rest/items*
  - Click the **Header** button under the URL, and add a header with *key* = *Accept*, and *Value* = ***application/json***
    - Just fill in the Key / Value fields in the GUI, and press the **+** sign
  - Make the request (click the **green arrows** to the right of the URL)
  - You should see a **JSON representation** returned
  - Change the Accept header value to ***application/xml***, and make the request again - you should see the **XML representation**
- Add an Accept header to *getAllItems()* specifying XML only

  *@RequestMapping(**headers={"Accept=application/xml"}**)*
  - **Restart** the server, and make the request with the rest-client specifying ***application/json***
  - You should see a **404** (Not Found) return - there is no controller matching the request URI and accept headers

[Optional] Additional Things to try

Lab

Tasks to Perform

- Modify *getAllItems()* to return the collection directly, instead of wrapping it in *ObjectWithList*, as shown below

  return results;

  – Make the request with the rest-client specifying *application/**xml***

  – You should see a **406** (Not Acceptable) status code

    - This means that the resource identified by the request is not capable of generating a response which is acceptable according to the accept headers

  – i.e. - There is a controller matching the request URI and accept headers

    - However, the controller can't generate the required response type, because the bare collection can't be mapped to XML

- That's it - you've worked with both XML and JSON representations

  – You've seen some of the subtleties of generating them, and of issues that might arise with various kinds of requests

STOP

LEARNINGPATTERNS

Client-Side Access to RESTful Service

REST Overview

REST and Spring MVC

Client-Side Access to RESTful Service

Common REST Patterns

REST Client Requirements

- Several standard requirements for any REST client, including:
  - **Create a connection** to the server hosting the services
  - **Set headers** for a request (e.g. Accept headers)
  - **Set URI Template Variables** / add request parameters
    - e.g. for the REST service */customers/{id}* you could start with a base URI of *http://foo.com/crm/rest/customers*
    - You need to add the id to the base - e.g. by concatenating a string
  - **Invoke** the REST service
  - **Extract** any results, possibly converting them to a Java object
- Can be done with standard Java classes
  - But it's tedious and lengthy
- Spring's ***RestTemplate*** takes care of the boilerplate code
  - Similarly to other templates, e.g. database access

RestTemplate Overview

- ***org.springframework.web.client.RestTemplate*** provides high level methods for invoking REST services
  - Handles connectivity requirements
  - Supports creation of URIs, including URI Templates
  - Converts to/from Java objects to other representations
    - e.g. XML or JSON
  - Supports all HTTP methods (GET, POST, PUT …)
- Below, *RestTemplate* accesses our customers service
  - ***RestTemplate.getForObject*** retrieves a customer by id
  - Note the URI Template (*{id}*) expanded from a value in the call (*"123"*)

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
Customer found = rt.getForObject(URI, Customer.class, "123");
```

getForObject() Details

- ***getForObject()*** performs a GET request for a Java object
  - Hence the name *getForObject*
- *getForObject()* signature (1 of 3 variations):

```
public <T> T getForObject(
    String url,                          // The request URL
    Class<T> responseType,               // Response type
    Object... urlVariables)              // URI Template values
  throws RestClientException
```

  - The ***url*** can include URI Template Variables
  - ***responseType*** is used to automatically convert the response
    - Uses HTTP message converters - as for the service side
  - ***urlVariables*** is used to expand the URI Template Variables
    - It is a varArgs value - see notes

Conventions for RestTemplate Methods

- Each HTTP method has corresponding *RestTemplate* methods
- Naming conventions indicate the HTTP method / return type
  - Consider ***getForObject()***
  - 1st part (***get***): Indicates HTTP GET is used
  - Last part (***Object***): Indicates return type is object
- Here is another get example (details covered later)

```
ResponseEntity<T> getForEntity( String url,
        Class<T> responseType, Object... urlVariables)
```

  - In this case, the response type is *org.springframework.http.**ResponseEntity***
  - It wraps the return object to allow access to HTTP the headers

RestTemplate Method Variations

- Each HTTP method is supported by many *RestTemplate* methods
  - Allowing for convenient usage under different situations
  - The *getForObject* variations are listed below
    - *url* and *responseType*, described previously, are not described again here

*T getForObject(**URI url, Class<T> responseType**)*

- Used when there are no URL Template vars to expand

*T getForObject(String url, Class<T> responseType, **Object... urlVariables**)*

- *urlVariables*: **varArgs** list of values to expand URL template vars
- Values are assigned by their order in the url

*T getForObject(String url, Class<T> responseType, **Map<String,?> urlVariables**)*

- *urlVariables*: **Map** of values to expand URL templates vars
- Consist of key (string) / value pairs - values are keyed by name

getForObject() Using Map

- In the example below, there is a URI Template with two variables
  - We use a map to supply the variable values
  - The map is name-based, and doesn't depend on the order of variables or values
  - Useful where multiple variables are present

```
    public static void main(String[] args) {
      RestTemplate rt = new RestTemplate();
      String URI =
       "http://foo.com/crm/rest/customers/fname/{fname}/lname/{lname}";
      Map<String,String> varsMap = new HashMap<String,String>();
      varsMap.put("fname",   "Jane");
      varsMap.put("lname", "Doe");
 Customer found = rt.getForObject(URI, Customer.class, varsMap);
}
```

Other RestTemplate Methods

- Examples of other *RestTemplate* methods include:

  void ***delete****(String url, Object... urlVariables)*

  –Delete the resources at the specified URI

  T ***postForObject****(String url, Object request, Class<T> responseType, Object... uriVariables)*

  –Create a new resource by POSTing the given object to the URI

  void ***put****(String url, Object request, Object... urlVariables)*

  –Create a resource by PUTting the given object to the URI

  –We'll cover some of these other methods in more detail later

LEARNINGPATTERNS

Lab 8.3 – A Client with RestTemplate

Lab – A Client with RestTemplate

Lab

- **Overview**: In this lab, we will write a client that uses *RestTemplate* to invoke a RESTful service
  - We'll use the services from our previous labs
  - We'll start simply, and in later labs look at more capability
- **Objectives**: Use RestTemplate to invoke a RESTful service
- **Builds on previous labs**: None
  - You will work in a new project and directory for this lab
- **Approximate Time**: 35-45 minutes

Lab Preparation

- New lab folder: **C:\StudentWork\Spring\workspace\Lab08.3**

Tasks to Perform

- Close all open files
- Create a new Java project called Lab08.3 in the workspace *
  - See Lab 1.1 instructions on how to do this
  - **Do not** switch to a Java Perspective if prompted
- Remember to **add the Spring user library** while creating the project as follows
  - **Click** *Next*, and in the Java Settings dialog, **click** the *Libraries* tab *
  - **Click** the *Add Library* button, and in the dialog that comes up, **select** *User Library* then **click** *Next*
  - **Check off** *Spring*, then **click** *Finish* in all open dialogs

Complete Client Code

Lab

Tasks to Perform

- Open *TestRest.java* for editing (package *com.javatunes.rest.client)*
  - Add the following code - look for the `// TODO` comments
- Look for the definition of *ID_URI*, and finish it by appending a URI Template variable for an item id to the *BASE_URI* variable
  - So we can call our REST service at a URI like *<webapp>/items/**2***
- In ***main()*** , do the following (see the TODO comments)
  - Create an instance of *RestTempate*
  - Use *RestTemplate.getForObject()* to get a *MusicItem*
    - Use *ID_URI* as the URI, *MusicItem.class* as the response type, and a single id value (a string) for the url variables
- **Run** the TestRest program and view the results
  - Right click on *TestRest.java*, select **Run As | Java Application**
  - You should see output showing your item in the console window
    - See notes about **viewing the correct console**

Complete Client Code

Lab

Tasks to Perform

- In *main()*'s next code section, use *getForObject()* to get a *MusicItem* as a string
  - Use a response type of *String.class*, and uri and uri variables as in the last call
- **Run** the TestRest program again and view the results
  - You should see output showing your item in the console window
  - It should show a JSON representation of the item
  - This is because RestTemplate is sending the following Accept header
    - Accept: text/plain, application/json, */*
  - Since json is listed, it is chosen as the representation
  - We'll see how to change this (if you want) later
- Next in *main()*, use a **map** to pass the URI variable's values
  - The other parameters should be the same as in the first call
- **Run** the program again and view the results

[Optional] Get a Collection

Lab

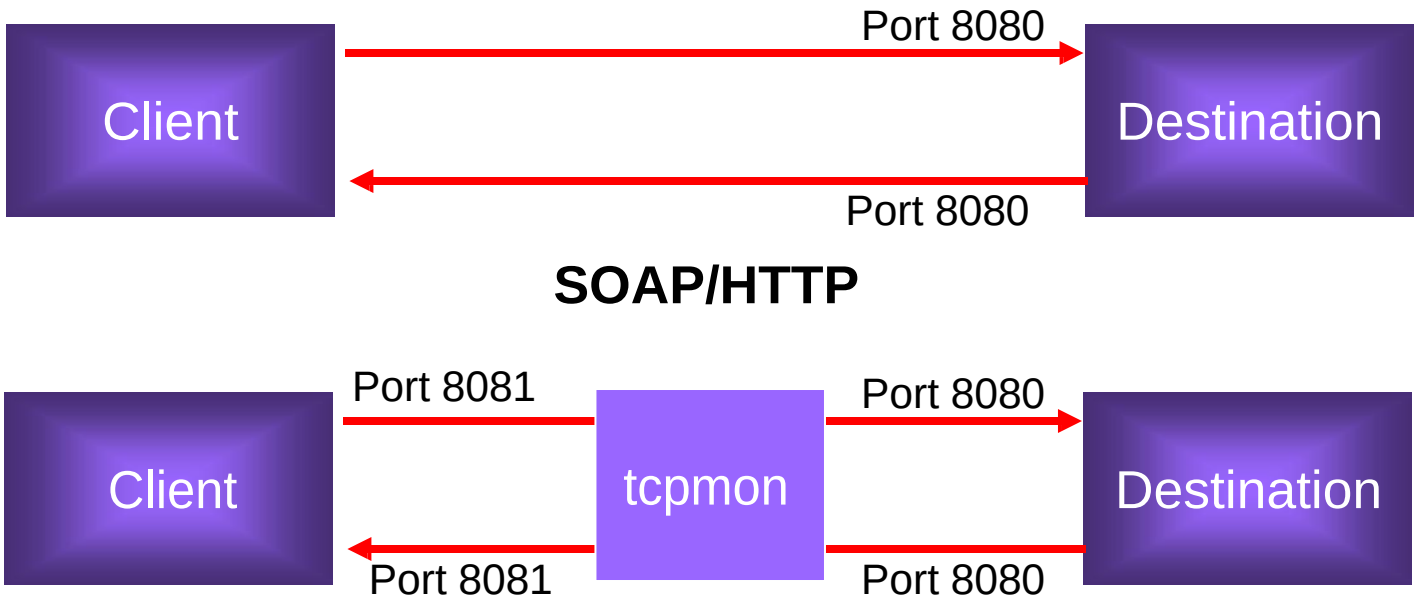
Tasks to Perform

- Back in your service, make sure that *getAllItems()* returns the bare list
  - Remove the setting of the *Accept* header to XML if you added that
  - The service should now return a JSON representation
- Back in the client, in the next code section, use *getForObject()* to get all the items
  - Use a URI of *BASE_URI*, and response type of *Collection.class*
  - No URL variable values are passed in (*BASE_URI* doesn't specify any URI Template variables)
- **Run** the TestRest program again and view the results
  - You should see a JSON string version of all the items in the collection
- You can see how easy it is to use *RestTemplate*
  - We'll go into more of its capabilities later

[Optional] tcpmon Overview

Lab

- tcpmon allows you to intercept TCP/IP messages
  - You simply give it a listen port and a target port, and it will sit in the middle of the requester and responder



[Optional] Use tcpmon

Lab

- You can optionally use tcpmon to view all the traffic between client and server
  - You'll see your REST services at work, and see what requests are being made, and what data is being transferred
  - This is not required, but useful to get a feel of how REST works

Tasks to Perform

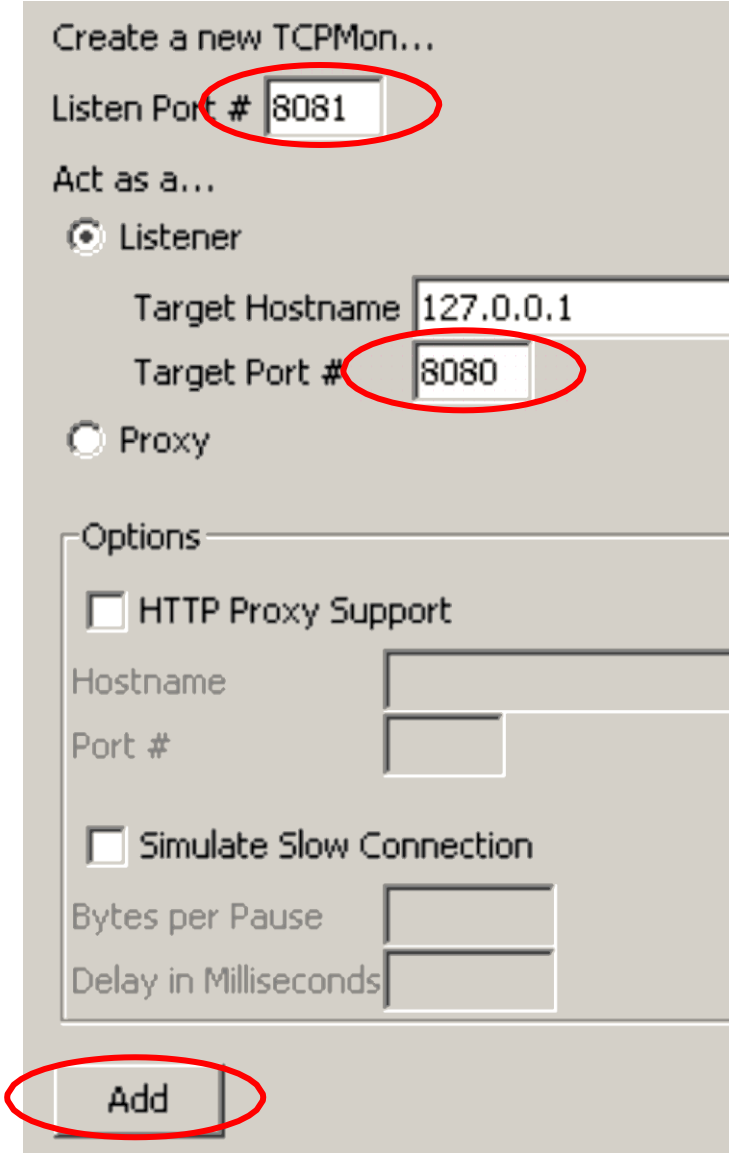
- Go to ***C:\StudentWork\SPRING\tcpmon***, and run ***tcpmon.bat***
  - You can double click on it - this starts up the tcpmon program
  - It opens in the **Sender** tab
  - Select the **Admin** tab, which we can use to intercept all the traffic between our REST client and the RESTful service
  - Continue as described on the next slide

[Optional] Use tcpmon

Lab

Tasks to Perform

- In the admin tab, fill in the values as follows:
  - Listen Port # **8081**
  - Target Port # **8080**
  - Click the [**Add**] button
- **Click the 8081 tab**, that appears (it shows request/response traffic)
- In your client, modify BASE_URI to use port 8081, instead of 8080
- Run your client
  - tcpmon should show the request/response traffic
  - See next slide for an example



[Optional] The TCPMon dialogue

Lab

State	Time	Request H...	Target Host	Request...
---	Most Recent	---	---	---
Done	2012-11-02 09:57:44	localhost	127.0.0.1	GET /javatunes/rest/items/1 HTTP/1.1

Remove Selected Remove All

```
GET /javatunes/rest/items/1 HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
User-Agent: Java/1.6.0_35
Host: 127.0.0.1:8081
Connection: keep-alive
```

REQUEST

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/xml
Transfer-Encoding: chunked
Date: Fri, 02 Nov 2012 13:57:44 GMT

fc
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<musicItem><artist>Annie Lennox</artist><id>1</id><listPrice>17.97</listPrice><
0
```

RESPONSE

STOP

© 2007-13 LearningPatterns Inc. All rights reserved.

[Optional] The TCPMon dialogue

Lab

RestTemplate.getForEntity()

- ***getForEntity()*** performs a GET request for an HTTP entity
  - Returns a ***ResponseEntity***, which extends ***HttpEntity***, and represents a request or response entity
  - *HttpEntity/ResponseEntity* allows access to the HTTP specific data of a response
- *getForEntity()* signature (1 of 3 variations):

```
<T> ResponseEntity<T> getForEntity(
    String  url,                    // The request URL
    Class<T>  responseType,         // Response type
    Object... urlVariables)         // URI Template values
  throws RestClientException
```

  - The arguments are as we've seen before
  - Details on the next page

HttpEntity and HttpHeaders

- *org.springframework.http.HttpEntity* / *ResponseEntity* represent a request or response entity
- Contains headers (in an *HttpHeaders* object) and a body
  - Access the headers and body via the methods below

```
public HttpHeaders getHeaders()
public T getBody()
```

- *ResponseEntity* adds the method *getStatusCode()*
  - Below is a simple example

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
ResponseEnity<Customer> foundEntity =
    rt.getForEntity(URI, Customer.class, "123");
System.out.println(foundEntity.getStatusCode());
```

Accessing Headers

- ***HttpHeaders*** has methods to access the headers
  - **Common headers** have specific methods, e.g.

long ***getContentLength()***: Content-length header value

MediaType ***getContentType()***: Content-type header value

  - **Generic access** useable for any header

String ***getFirst(String headerName)***: First value for header

  - Many others - see the documentation
  - Below is a simple example of accessing headers

```
    // Assume same URI and RestTemplate object as before ...

ResponseEnity<Customer> foundEntity =
    rt.getForEntity(URI, Customer.class, "123");
System.out.println("getForEntity has content type: " +
                        foundEntity.getHeaders().getContentType());
```

RestTemplate.exchange()

- *RestTemplate.exchange()* takes an *HttpEntity* object
  - Can be used to set headers for the request
  - The signature is below

```
<T> ResponseEntity<T> exchange(
    String url,                    // The request URL
    HttpMethod method,             // HTTP method e.g. GET
    HttpEntity<?> requestEntity,   // Request entity
    Class<T> responseType, Object...urlVariables)
  throws RestClientException
```

- *url*, *responseType*, *urlVariables* as seen earlier
- ***method*** is the HTTP method to use - GET, POST, etc.
- ***requestEntity*** is the entity to write to the request
  - May include headers

Setting Headers on a Request

- Below, *exchange()* is used to set an accept header specifying JSON
  - Using the enum *org.springframework.http.**MediaType***
  - Also shows some of the API for the types used (see notes)
  - It's somewhat lengthy, but fairly straightforward

```
// Assume same URI and RestTemplate object as before ...

List<MediaType> accepts = new ArrayList<MediaType>();
accepts.add(MediaType.APPLICATION_JSON);
HttpHeaders headers =new HttpHeaders();
headers.setAccept(accepts);
HttpEntity<customer>requestEntity =
        new HttpEntity<Customer>(headers);
ResponseEntity<Customer> exchangeEnt = rt.exchange(ID_URI,
        HttpMethod.GET, requestEntity, Customer.class,"123");
System.out.println(exchangeEnt.getHeaders().getContentType());
```

Error Handling

- A number of choices in handling errors on the server side
- For example, if lookup of an item by id finds no item you can:
  - Throw an exception in controller (e.g. IllegalArgumentException)
  - Return null
    - Both of these will **result in a (runtime) exception** on the client side
  - Provide a way to test ahead of time if a call will fail
    - e.g. make sure that the id is found by the REST service below
- You must decide how to deal with these in the client
  - e.g. handling exceptions, or calling a test method

```
// Based on REST controller used before

@RequestMapping(value="/exists/{id}", method=RequestMethod.GET)
@ResponseBody
public Boolean checkForCustomer(@PathVariable("id") Long id)
{ /* Check for customer with given id */ }
```

Summary of RestTemplate Usage

- We've looked at some core functionality of *RestTemplate*
  - Focusing on using get to make requests
- We've seen there are many variants of each capability
  - Multiple versions of each method like *getForObject()*
  - Multiple methods to accomplish a particular task
    - e.g. *getForObject()*, *getForEntity()*, *exchange()*
- There is broad HTTP method support
  - Methods to do POST, PUT, DELETE, etc.
  - Multiple variants of each method
  - Other ways to execute requests (e.g. the *execute()* method)
- We'll cover some more of this later
  - There are many details beyond the scope of the course
  - Look at the documentation !!

LEARNINGPATTERNS

Lab 8.4 – [Optional] Accessing Headers

Lab – Accessing Headers

Lab

- **Overview**: In this lab, we'll use additional *RestTemplate* functionality to access the HTTP headers
  - We'll examine the headers from a *getForEntity()* request
  - We'll use *exchange()* to set accept headers
- **Objectives**: Use *RestTemplate* to get and set headers when accessing a RESTful service
- **Builds on previous labs**: 8.3
  - You will continue to work in your Lab08.3 project
- **Approximate Time**: 25-45 minutes (depending on optional parts done)

Use getForEntity()

Lab

Tasks to Perform

- Open *TestRest.java* for editing
  - Comment out all the *getForObject()* invocation and output code
- Add code to do the following
  - Use *RestTemplate.getForEntity()* to get a *MusicItem*
    - Use *ID_URI* as the URI, *MusicItem.class* as the response type, and a single id value (a string) for the url variables
    - The return type of the call will be *ResponseEntity<MusicItem>*
    - Access and output the content type and status code from the return value
    - Access and output the body from the return value
- **Run** the *TestRest* program and view the results
  - The returned item should be the same
  - You should also be able to see the content type and status codes of the call

[Optional] Set Headers

Lab

Tasks to Perform

- Time permitting, do the following
- Create an *HttpHeaders* object
- Create an *ArraryList<MediaType>*
  - Add in *MediaType.APPLICATION_XML* to the list
  - Set the accepts list of the headers to be your list of *MediaType*
- Create an *HttpEntity<MusicItem>*- passing in your headers object to the constructor
- Call *RestTemplate.exchange()*, to GET an item by id
  - Pass in your *HttpEntity* as the request entity in the call
  - The *HttpMethod* should be *HttpMethod.GET*
  - The response type should be *MusicItem.class*, and other parameters as in previous calls
  - Access and output the content type and body from the return value
- **Run** the program - look at the results

[Optional] Change Media Type

Lab

Tasks to Perform

- Change the media type in your list of media type to MediaType.APPLICATION_JSON
  - Comment out the code that adds the XML type, and add in the JSON
  - **Run** your program again - notice that the content type of the exchange call is now application/json
- If you want to see the json that was returned, do the following
  - Create an *HttpEntity<String>* passing in your same headers object to the constructor
  - Call *exchange()* as before, except use your new request entity, and use *String.class* for the response type
  - Get the body from the return value of *exchange()* and output it
  - **Run** the program, and view the output
- That's it - you've used some of the entity capabilities
  - We'll explore more *RestTemplate* capabilities later

STOP

LEARNINGPATTERNS

Common REST Patterns

REST Overview

REST and Spring MVC

Client-Side Access to RESTful Service

Common REST Patterns

The REST Methods

- We'll now review some standard REST methods (or verbs)
  - We'll look at the following details for each:
  - The **REST method** and its effect
    - For GET, POST, PUT and DELETE
  - The **CRUD equivalent operation** (assuming a standard Create-Read-Update-Delete) system
  - The **kind of entity** it operates on (the noun)
    - Either a single entity or a collection of entities
  - Other characteristics, such as is it **idempotent** or **cacheable**
    - An idempotent operation can be applied multiple times without changing the result
  - A **Controller implementation** example (the server side)
    - We won't show the business logic for the examples
  - A ***RestTemplate* client** implementation example (a client)

GET: Retrieve Information

- **REST Verb**: GET (We've seen many examples of this already)
- **CRUD Equivalent**: Read
- **Noun**: Collection URI (multiple) or Entity URI (single)
- **Idempotent**: So it should not initiate state change
- **Cacheable**
- Examples (only pertinent parts of URL shown in all examples)
  - Multiple: GET /**customers**
  - Single: GET /**customers/123**

```
// Controller - Assumes @RequestMapping("/customers") on class

@RequestMapping // Multiple: Match URI like /customers
@ResponseBody
public Object getAll() {...} // See notes about return value

@RequestMapping("/{id}") // Single: Match URI like /customers/123
@ResponseBody
public Customer getCustomer(@PathVariable Long id) { ... }
```

POST: Add New Information

- **REST Verb**: POST
- **CRUD Equivalent**: Create
- **Noun**: Collection URI (Since entity URI only available after creation)
- Example: POST /**customers**
- **Server**: To implement this specify HTTP POST on the handler
  - Done via *@RequestMapping*'s ***method*** element (See next slide also)

```
// Controller - Assumes @RequestMapping("/customers") on class
// Must specify GET now (see notes)
@RequestMapping(method=RequestMethod.GET)
@ResponseBody
public Object getAll() {...}

@RequestMapping(method=RequestMethod.POST) // Matches on POST only
   @ResponseStatus(HttpStatus.CREATED)
   @ResponseBody
   public void createCustomer(@RequestBody Customer cust) {
      // Create a Customer from the passed in parameters
   }
```

Server: POST Controller - Details

- *@RequestMapping* specifies the HTTP method as POST
  - Via ***method=RequestMethod.POST***
  - Note that *getAll()* must now specify that it handles GET
    - Otherwise, it would also be available to handle a POST
- *createCustomer()* receives a Customer object *
  - Created from the request body by an Http message converter
  - ***@RequestBody*** binds the *cust* method parameter to the request body
- Note the setting of the CREATED status code
  - We would also return the created object in this example

```
@RequestMapping(method=RequestMethod.POST) // Matches on POST only
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody
public Customer createCustomer(@RequestBody Customer cust)
{ /* … */ }
```

Server: Setting Location Header for POST

- Common practice for a controller to **set the location header**
  - So the client can request just a **location URI** instead of an object
    - Which it may use for later access to the new object
- Below, we show this in a handler
  - The handler method has an *HttpServletResponse* parameter
    - Which is initialized with the current response object (see notes)
  - This is used to set the location header

```
@RequestMapping(method=RequestMethod.POST) // Matches on POST only
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody
public Customer createCustomer(@RequestBody Customer cust,
                               HttpServletResponse response) {
  Customer created = // Create customer - detail not shown
  response.setHeader("Location", "/customers/" + created.getId());
  return created;
}
```

Client: RestTemplate for POST

- *RestTemplate.**postXXX()*** can be used to send a POST
  - Creates a new object using data sent in the call
- *postForObject()* signature (1 of 3 variations):
  - POSTs to the specified URI with the given object

```
<T> T postForObject (
    String url,                    // The request URL
    Object request,                // Object to create
    Class<T> responseType)         // Type of return value
  throws RestClientException
```

- Below is an example usage

```
String URI = "http://foo.com/crm/rest/customers";
RestTemplate rt = new RestTemplate();
Customer custData = new Customer("Jane", "Doe");
Customer created = rt.post(URI, custData, Customer.class);
```

PUT: Update Information

- **REST Verb**: PUT
- **CRUD Equivalent**: Update
- **Noun**: Entity URI (URI of entity that is updated)
- Example: PUT /**customers/123**
  - URI includes the id - remaining data is in the request body
- **Server**: Specify HTTP PUT on the handler - Similar to POST
  - *updateCustomer()*, below, updates the passed in customer
  - We return nothing - can also return the updated object if you want

```
// Implementation - Assumes @RequestMapping("/customers") on class

  // Matches URI like /customers/123 with method of PUT (only)
  @RequestMapping(value="/{id}",  method=RequestMethod.PUT)
  @ResponseBody
  public void updateCustomer(@PathVariable("id") Long id,
                             @RequestBody Customer cust) {
    // Update Customer with given id using values in Customer object
  }
```

Client: RestTemplate for PUT

- *RestTemplate.**put()*** can be used to send a PUT
  - Must send the object to update in the call
- PUT method signature (1 of 3 variations):
  - PUT to the specified URI with the given object

```
<T> T put (
    String url,                    // The request URL
    Object request,                // Object to update
    Object... urlVariables)        //      URI Template
            values
  throws RestClientException
```

- Below, we update the customer retrieved for id = 123

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
Customer found = rt.getForObject(URI, Customer.class, "123");
found.setFirstName("NewName");
```

Client: RestTemplate for PUT

```
rt.put(URI, found, "123");
```

DELETE: Remove an Entity

- **REST Verb**: DELETE
- **CRUD Equivalent**: Delete
- **Noun**: Entity URI (URI of entity that is deleted)
- Example: DELETE /**customers/123**
- **Server**: Below we show a controller annotated for DELETE

```
// Implementation - Assumes @RequestMapping("/customers") on class

  // Matches URI like /customers/123 with method of DELETE (only)
  @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
  @ResponseStatus(HttpStatus.NO_CONTENT)
  @ResponseBody
  public void deleteCustomer(@PathVariable("id") Long id){
    // Delete customer with given id
  }
```

Client: RestTemplate for DELETE

- *RestTemplate.**delete()*** can be used to send a DELETE
- *delete()* signature (1 of 3 variations):
  - Deletes the resource at the specified URI

```
public void delete (
    String url,                          // The request URL
    Object... urlVariables)              // URI Template values
  throws RestClientException
```

- Below, we delete the customer with id = 123

```
String URI = "http://foo.com/crm/rest/customers/{id}";
RestTemplate rt = new RestTemplate();
rt.delete(URI, "123");
```

LEARNINGPATTERNS

[Optional] Lab 8.5: More REST Operations

Lab 8.5 – More REST Operations

Lab

- **Overview**: In this (optional) lab, we will implement additional REST operations on the server side
  - PUT, DELETE, and POST
    - Most of the code will be given to you - you'll just need to add the code to turn them into RESTful services
  - We'll also write clients for these using *RestTemplate*
- **Objectives**:
  - Use additional REST capabilities
- **Builds on previous labs**: none
- **Approximate Time**: 45-60 minutes

Lab Preparation

Lab

- There are two new directories where you will do your work:

  Server: C:\StudentWork\Spring\workspace**\Lab08.5**

  Client: C:\StudentWork\Spring\workspace**\Lab08.5-Client**

Tasks to Perform: Server Project

- Remove any projects from the server, close all open files\projects
- Create a new Dynamic Web project called Lab08.5
  - See Lab 6.1 instructions if you need more detail on this
  - Remember to set the context root to javatunes
  - Remember to add the Spring user library to the build path and also via the **Deployment Assembly** (see notes)
- **Create a Java project** called **Lab08.5-Client** in the workspace *
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring User Library**
  - **Do not** switch to a Java Perspective if prompted

Finish Controller for DELETE

Lab

Tasks to Perform

- In the **Lab08.5** project (the server) open *ItemsController.java* for editing, find the *deleteItem()* method, and add the following to it
  - *@RequestMapping* that specifies the correct URI and HTTP method
    - The URI will be something like *<webapp>/items/**2***
    - The "2" part will be represented by an URI Template variable, and your controller method will extract this as the id of the item to delete
    - The method is HTTP DELETE
  - *@ResponseBody* to specify the return value is used to create response
  - *@PathVariable* to bind the URI Template variable to the id parameter
  - Add *@ResponseStatus* to specify NO_CONTENT status
- **Add** the project to the server (see earlier labs if you need details)
  - **Restart** the server

Finish Client for DELETE

Lab

Tasks to Perform

- In the **Lab08.5-Client** project open *TestRest.java* for editing
  - Look for the TODO comments in *main()* that include DELETE
- Add a call to the RESTful DELETE service using *RestTemplate*
  - Use one of the *RestTemplate.delete()* variants
  - Use the simplest method that you can here and in the rest of the lab
    - This is generally the one that takes an Object… argument if you need to pass in URI Template variable values
  - Use a URI with a URI Template variable for the id
- **Run** the client / view the results - the item you deleted shouldn't be present in the collection of all items fetched after the delete
  - **Note**: See notes about running the client more than once

Finish and Test PUT

Lab

Tasks to Perform

- In *ItemsController.java*, find the *updateItem()* method, and add:
  - *@RequestMapping* that specifies the correct URI and HTTP method
    - The URI will be something like *<webapp>/items/2* (with a URI Template var)
    - The method should be an HTTP PUT
  - *@ResponseBody* to specify the return value is used to create response
  - *@PathVariable* to bind the URI Template variable to the *id* parameter
  - *@RequestBody* to bind the request body to the *item* parameter
  - **Restart** the server
- In *TestRest.java*, add the following (Look for TODO comment with PUT)
  - Add a call to the RESTful PUT service using *RestTemplate*
  - The URI should include a URI Template var for the id
  - Use the *putId* variable as your id value, and the found (and changed) object we already include in the code as the request object
  - **Run** the client / view the results - the 2nd GET should show new values

Finish Controller for POST

Lab

Tasks to Perform

- In *ItemsController.java*, find the *createItem()* method, and add:
  - *@RequestMapping* that specifies the correct URI and HTTP method
    - The URI will be something like <webapp>/items
    - There is NO URI Template variable in the URI
    - The method should be an HTTP POST
  - *@ResponseBody* to specify the return value is used to create response
  - *@ResponseStatus* to specify CREATED status
  - *@RequestBody* to bind the request body to the *item* parameter
    - This will be in the method parameter list
  - Within the *createItem()* method, set the ***Location*** header via the *HttpResponse* parameter passed into the method
    - The value will be of the form */items/idValue*
- **Restart** the server

Finish Client for POST

Lab

Tasks to Perform

- In *TestRest.java*, add the following (Look for TODO comment w/POST)
  - Add a call to the RESTful POST service using *RestTemplate*
  - The URI should include a URI Template var for the id
  - Use the already created item (*newItem*) supplied in the code as the request object
  - First use *postForObject()* to create and get the object back
  - Next, use *postForLocation()* to create and get just the location back
    - This second call creates a new object (with a different id) using the same data as in the *postForObject()* call - that's fine for our testing
- **Run** the client / view the results
  - You should see the newly created item output, and then a location URI(with a different id)
- That's it - you've written and used 3 new REST services

STOP

Review Questions

- What is REST? What are RESTful services?
- What representations does REST support for data?
- How do you implement RESTful services using Spring?
- How do you create RESTful clients with Spring?

Lesson Summary

- **REST** (**Re**presentational **S**tate **T**ransfer) is a set of principles on defining and accessing resources on a network
  - REST services are stateless, and systems communicate by transferring representations of resources
  - RESTful resources are uniquely identified by URI, and can include URI Templates
  - REST uses only a few methods based on HTTP
    - GET, POST, PUT, DELETE

- REST can support many data representations
  - Including text, XML, JSON, and others

Lesson Summary

- Spring's REST support builds on top of Spring MVC, including:
  - *@RequestMapping* maps RESTful URIs to standard controllers
  - *@PathVariable* supports URI templates
  - *@RequestBody* binds incoming HTTP data to Java objects
  - *@ResponseBody* binds return values to the response body
  - HTTP Message converters support different data representations

- Spring's RestTemplate takes care of much of the boilerplate code for REST clients, including:
  - Handling connectivity requirements
  - Supporting creation of URIs, including URI Templates
  - Converting to/from Java objects to other representations e.g. XML or JSON
  - Supporting all HTTP methods (GET, POST, PUT …)

LEARNINGPATTERNS

[Optional] Module 9: Aspect Oriented Programming

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

Lesson Objectives

- Understand what Aspect Oriented Programming (AOP) is and the kinds of issues that it addresses
- Define Spring AOP using @AspectJ annotations
- Define aspects, pointcuts, and advice
- Learn about the Spring AOP architecture
- Understand and use Pointcut expressions and the different kinds of advice in Spring AOP
- Learn how to define Spring AOP using its XML schema support
- Be aware of the strengths and weaknesses of AOP

© 2007-13 LearningPatterns Inc. All rights reserved.

Overview

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

The Issue with Crosscutting Concerns

- **Crosscutting concerns**: Concerns (functionality) extending across multiple modules
  - For example, logging and transaction control
  - Generally present in any complex system
- Easy to conceptualize directly
  - Can include the required code where needed
  - For example, using logging statements or transactional control statements in the normal code flow
- Direct inclusion of crosscutting code can increase complexity
  - Making systems difficult to understand and maintain
  - Core functionality is interweaved with and complicated by crosscutting code
- **AOP** is designed to address this issue

Crosscutting Illustrated

- Below, are two code segments
  - The 1st without crosscutting concerns, the 2nd with
  - Note the added complexity of the 2nd sample

```
void transfer(Account fromAccount, Account toAccount, int amount) {
   fromAccount.withdraw(amount);
   toAccount.deposit(amount);
}
```

```
void transfer(Account fromAccount, Account toAccount, int amount) {
   Transaction tx = database.newTransaction();
   try {
       fromAccount.withdraw(amount);
       toAccount.deposit(amount);
       tx.commit();
       systemLog.log("Transfer",   ...   );
   }
   catch(Exception e) {
       tx.rollback();
       throw e;
   }
}
```

Aspect Oriented Programming (AOP) Defined

- AOP **encapsulates crosscutting concerns** in one place called **aspects**
  - **Separates** aspects from other system components
  - Aspects are **composed** with other components to produce the needed functionality
- Key AOP concepts:
  - **Aspect**: A module holding cross-cutting concerns (a class in Spring)
  - **Advice**: The action taken by an aspect (its functionality)
  - **Joinpoint**: A point during the execution of a program, such as a method call, where advice may run
  - **Pointcut**: A specification that matches joinpoints. Advice runs at any joinpoint that matches an associated pointcut
  - **Advised (target) object**: An object being advised by an aspect
  - **Weaving**: The process of combining the aspects with the rest of your application to create advised objects

Spring AOP Introduction

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

Spring AOP With AspectJ Annotations

- Spring AOP can be defined with annotations and/or XML
  - We'll cover the annotation based usage first
- Spring uses **@AspectJ annotations** to program AOP
  - AspectJ is an open source AOP framework for Java AOP
  - Uses Java classes annotated with special annotations
- This section provides a simple example of Spring AOP using @AspectJ notations
  - It leaves out a lot of detail, and focuses on the basic concepts and syntax
  - We will cover some more of the detail later

Defining an Aspect with @AspectJ

- We'll use our *JavaInstructor* class as an example
  - Focusing on its *teach()* method
  - Assume we want to "set up a display projector" before any call to *teach()*
  - We'll implement this as AOP advice
- We'll code this using the *TeachingSupport* aspect below
  - **@Aspect** indicates that the class is an aspect
  - It doesn't do anything yet - that comes next

```
package com.javatunes.teach;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class TeachingSupport {  // Doesn't really do anything yet
}
```

Defining a Pointcut

- A **joinpoint** is a location in your program
  - This is where your advice can run
  - In Spring AOP, this is basically a method
- A **pointcut** is a specification that matches joinpoints
  - Pointcuts are used to select the joinpoints where advice will run
- ***@Pointcut*** and an expression to specify the joinpoint are used to define a pointcut
  - ***@Pointcut*** applies to methods (only) in Spring
    - Having signature *public void*
    - The method name becomes the name of the pointcut
  - An **AspectJ expression** specifies the joinpoints affected

Defining a Pointcut

- Below we specify a pointcut that matches "**any execution of a method called teach**"
  - The pointcut is called ***anyTeach()***, the same as the method
  - The AspectJ expression is ***execution(* teach(..))***
  - We'll go into more detail on the expression later
  - Note that there is no code nor return value for the method

```
package com.javatunes.teach;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TeachingSupport {
  @Pointcut("execution(* teach(..))")
  public void anyTeach() {} // Shouldn't be any body in this method
}
```

Defining Advice

- **Advice** is associated with a pointcut expression
  - It runs when the pointcut expression is matched
  - Advice is specified by an **annotation** (there are several) on a method
    - It includes a **pointcut** name specifying when the advice is run
    - The annotated method defines the action of the advice
- Below, ***@Before*** declares advice in ***setupDisplayProjector()***
  - It will run **before** any method execution matching the ***anyTeach()*** pointcut

```
@Aspect // note - package statement, imports omitted
public class TeachingSupport {
   @Pointcut("execution(* teach(..))") // syntax covered later
   public void anyTeach() {}

   @Before("anyTeach()")
   public void setupDisplayProjector() {
      System.out.println("Turning On Display Projector");
   }
}
```

Configuring Spring

- To use @AspectJ aspects, you must enable support for it
  - Done in the Spring configuration via the following element:

```
<aop:aspectj-autoproxy/>
```

  - This enables **autoproxying** - where Spring generates a proxy for beans that use AOP when the bean is looked up

- Aspect classes and beans using AOP must be declared in the Spring configuration, as shown below
  - They must be managed by the Spring container to enable AOP

```
<beans> <!-- xmlns and schema locations omitted - see notes -->

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

  <aop:aspectj-autoproxy/>
</beans>
```

Triggering Advice

- When the code at bottom is run – guess what happens?
  - **The advice is executed** and the following output results:

```
Turning On Display Projector
Spring is cool
```

  - The *teacher.teach()* call matches the *anyTeach()* pointcut
    - Which is defined in the *TeachingSupport* aspect
  - The before advice (the *setupDisplayProjector()* method) therefore runs before the method executes

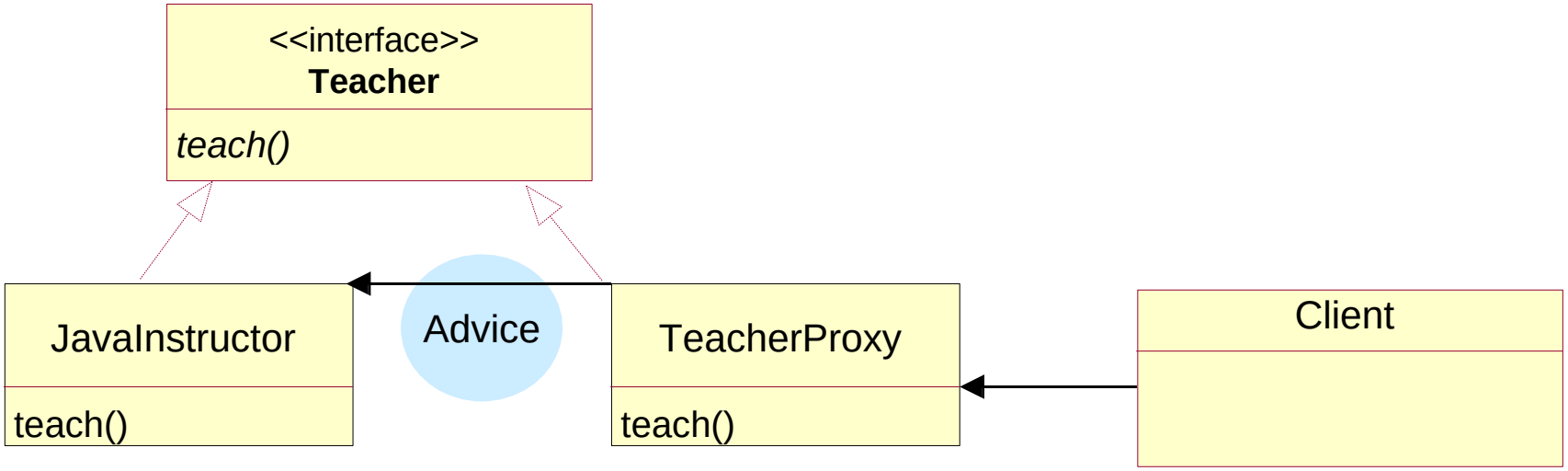
- Spring automatically generated any needed AOP code
  - *JavaInstructor* and *Teacher* **didn't need to change** !

```
ClassPathXmlApplicationContext ctx =
 new ClassPathXmlApplicationContext("applicationContext.xml");
Teacher teacher = (Teacher)ctx.getBean("springGuru");
teacher.teach();
```

More on How Spring AOP Works

- Spring uses proxies (by default) to implement the AOP functionality
  - When you look up an advised bean, you receive a proxy instead
  - When you invoke on the proxy, it adds advice applicable to the target

- **Spring AOP only supports method joinpoints**
  - Since it is proxy based, it can't support joinpoints on things like field access and constructors



LEARNINGPATTERNS

Lab 9.1 – Hello AOP World

Lab 9.1 – Using Spring AOP

Lab

- **Overview**: In this lab, we will work with Spring AOP
  - We'll use the @AspectJ annotations, and create a very simple system that shows the basics of AOP
  - Notice that this lab builds on the types used in our very early Spring labs – *JavaInstructor* and its associated types
- **Objectives**:
  - Become familiar with Spring AOP using @AspectJ annotations, and use it in a program
- **Builds on previous labs**: None
- **Approximate Time**: 25-45 minutes (depending on optional parts)

Lab Preparation

- New lab folder: **C:\StudentWork\Spring\workspace\Lab09.1**

Tasks to Perform

- Close all open files
- Create a new Java project called Lab09.1 in the workspace *
  - See Lab 1.1 instructions on how to do this
  - **Do not** switch to a Java Perspective if prompted
- Remember to **add the Spring user library** while creating the project as follows
  - **Click** *Next*, and in the Java Settings dialog, **click** the *Libraries* tab *
  - **Click** the *Add Library* button, and in the dialog that comes up, **select** *User Library* then **click** *Next*
  - **Check off** *Spring*, then **click** *Finish* in all open dialogs

Defining the Aspect (Pointcut/Advice)

Lab

Tasks to Perform

- Open up the file **TeachingSupport.java**
  - Declare this class to be an aspect using @Aspect
- Define a pointcut in *TeachingSupport*, using ***@Pointcut***
  - Use the following pointcut expression to specify any execution of the teach method (See the manual slides for sample code):

    execution(* teach(..))
  - Name the pointcut ***anyTeach()*** (by associating *@Pointcut* with a method of this name)
- Define advice in *TeachingSupport* using ***@Before***
  - Associate it with the *anyTeach()* pointcut
  - Define it on the method ***setupDisplayProjector()***
  - The method can print something using *System.out.println()*
  - That's all we need in our aspect right now

Configuring the Aspect

Lab

Tasks to Perform

- Open up the file *src/applicationContext.xml*
  - Notice the *aop* namespace declarations in the <*beans*> element
  - Add in the following element: **<aop:aspectj-autoproxy/>**
  - Add in a new bean element for *TeachingSupport*
    - The id does not matter
  - These are both **required** for Spring to recognize the aspect
- **Open** up the file ***TeachMeSpring.java*** and review it
  - All it does is look up a *JavaInstructor* and call *teach()*
- **Run** the program in ***TeachMeSpring.java***
  - You should see the output from your advice
  - If your advice isn't run, make sure the joinpoint expression in the Pointcut is exactly as in the manual – make sure all parenthesis and quotes are balanced

[Optional] Additional Pointcut & Advice

Lab

[Optional] Tasks to Perform

- Review the *JavaInstructor*, *InfoSource*, and *SpringCourseBook* classes to refresh your memory
  - We'll define advice to check the data in an *InfoSource*
- In the *TeachingSupport* class, declare an additional pointcut
  - Use the following pointcut expression to specify an execution of any method in the *InfoSource* interface *

    execution(* com.javatunes.teach.InfoSource.*(..))
  - Associate the pointcut with a method called *inInfoSource()*
- Define additional advice in *TeachingSupport* using *@Before*
  - Associate it with the *inInfoSource()* pointcut
  - Define it on the method ***checkData()***
  - The method can print something using *System.out.println()*
- Build/run your program
  - You should see the output from your additional advice

[Optional] Additional Things to Try

Lab

[Optional] Tasks to Perform

- In *TeachMeSpring.java* try adding code that creates a JavaInstructor, rather than getting it from the Spring container
  - See example code below
- Run your program
  - What happens? Is advice called when the teach() method is called on this instance?
  - Why or why not?

```
JavaInstructor ji = new JavaInstructor();
ji.setInfo(new SpringCourseBook());
System.out.println("*** Teaching with new-ed instructor ***");
ji.teach();
```

STOP

LEARNINGPATTERNS

Pointcut Expressions and Advice

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

Pointcut Expressions

- Pointcuts are defined using AspectJ expressions
  - The expressions include **designators** that specify the joinpoint
  - We'll cover these enough so the concepts are clear
  - We won't cover the many details - they are in the AspectJ docs
- The ***execution*** designator has the general form shown at bottom
  - The **return type**, **name**, and **parameter** patterns are required
  - All others are optional
  - Wildcards (e.g. *) are allowed in the patterns
  - We'll review several examples to show how it works

```
execution(modifiers-pattern? ret-type-pattern
          declaring-type-pattern? name-pattern(param-pattern)
          throws-pattern?)
```

Sample execution Designator Patterns

- ***execution(public * *(..))***: Matches any public method
  - *public* = the modifier pattern
  - The 1st * = the return type, and matches any type
  - The 2nd * = the method name and matches any name
  - *(..)* matches "any number of parameters" (zero or more)
- ***execution(* teach(..))***: Matches any invocation of *teach()*
  - * = the return type (any type)
  - *teach* = the method name, and matches teach only
  - *(..)* Matches "any number of parameters" (zero or more)
- ***execution(* com.javatunes.teach.InfoSource.*(..))***: Matches invocation of method defined in the *InfoSource* interface
  - * = the return type (any type)
  - *com.javatunes.teach.InfoSource.** = the method name, and matches any method in the *InfoSource* interface in the given package
  - *(..)* Matches "any number of parameters" (zero or more)

Sample execution Designator Patterns

- ***execution(* com.javatunes.teach.*.*(..))***: Matches any method invocation from types in package *com.javatunes.teach*
  - *com.javatunes.teach.*.** = the method name – the ***.*** means any method in any class
- ***execution(* teach())***: Matches any invocation of *teach()* (from any class) that is invoked with no arguments
  - *()* Matches "zero parameters"
- ***execution(* teach(String))***: Matches any invocation of *teach()* that takes a String argument
  - *(String)* Matches "a single String argument"
- ***execution(* teach(*, String))***: Matches any invocation of *teach()* invoked with two arguments, with the 2nd being a *String*
  - *(*, String)* Matches "any param, followed by a *String* param"
- ***execution(* Teacher.teach(..))***: Matches any invocation of *Teacher.teach()* that is invoked with any number of arguments

Other Designators Available in Spring AOP

- ***within***: Limits matching to joinpoints within certain types
- ***this***: Limits matching to joinpoints where the bean reference (Spring AOP proxy) is an instance of the given type
- ***target***: Limits matching to joinpoints where the target object (application object being proxied) is an instance of the given type
- ***args***: Limits matching to joinpoints where the arguments are instances of the given types
- ***@target***: Limits matching to joinpoints where the class of the executing object has an annotation of the given type
- ***@args***: Limits matching to joinpoints where the runtime type of the actual arguments passed have annotations of the given type(s)
- ***@within***: Limits matching to joinpoints within types that have the given annotation
- ***@annotation***: Limits matching to joinpoints where the subject of the joinpoint has the given annotation

Sample Designator Patterns

- ***within(com.javatunes.teach.Teacher+)***: Matches invocation on any object of type Teacher or its subclasses
  - + means include subclasses – needed as *Teacher* is an interface *
- ***within(com.javatunes.teach.*)***: Matches invocation on any type in the single package *com.javatunes.teach*
- ***within(com.javatunes.teach..*)***: Matches invocation on any type in the package *com.javatunes.teach* or its subpackages
  - .. mean include subpackages
- ***target(com.javatunes.teach.Teacher)***: Match invocation where the target object (not the proxy) is a *Teacher* instance
- ***args(String)***: Match invocation on any method with a String arg
- ***args(*)***: Match on method with a single arg of any type
- ***args(*, String)***: Match on method with two args – the first of any type, the second of type String

Combining Pointcut Expressions

- Combine pointcut expressions using **&&** (and), **||** (or) and **!** (not)
  - You can use explicit pointcut expressions, or named pointcuts
- Assume the pointcut *anyTeach()* is defined in the aspect *School*
  - The *@Before* in *TeachingSupport* "ands" it with *targetTeacher()*
  - The advice will **only** run on a teach method executed on a *Teacher*

```
@Aspect
public class School {
    @Pointcut("execution(*  teach(..))")
    public void anyTeach() {}
}
```

```
@Aspect
public class TeachingSupport {
   @Pointcut("target(com.javatunes.teach.Teacher)")
   public void targetTeacher() {}

   @Before("targetTeacher() && School.anyTeach()")
   public void setupDisplayProjector() {
      System.out.println("Turning On Display Projector");
   }
}
```

Kinds of Advice

- Spring AOP supports several different advice types
  - They differ in when the advice is executed
  - We describe them below (labeled with their @AspectJ names)

- ***@Before***: Runs before each joinpoint
- ***@AfterReturning***: Runs after each joinpoint returning normally
- ***@AfterThrowing***: Runs after each joinpoint that throws a Throwable
- ***@After***: Runs after each joinpoint regardless of how it completes
- ***@Around***: Runs in place of each joinpoint
  - You can execute the joinpoint by calling *proceed()*

- The kind of advice you use controls when the advice is executed
  - For example, *@After* advice will execute code no matter how the joinpoint completes - i.e. completes normally or throws an exception

Advice Examples

- In the example below, the *bringTeacherADrink()* advice will run after any invocation of *teach()*
  - Even if *teach()* threw an exception
  - The *getEvals()* advice only runs if *teach()* completes normally

```
@Aspect
public class TeachingSupport {
   @Pointcut("execution(* teach(..))")
   public void anyTeach() {}

  @AfterReturning("anyTeach()")
   public void getEvals() {
      System.out.println("Hmm - How did the teach go?");
   }

   @After("anyTeach()")
   public void bringTeacherADrink() {
      System.out.println("Ahhh, Bringing teacher a drink");
   }
}
```

Around Example

- **Around** advice actually **replaces the running of the joinpoint**
  - You can execute advice before the joinpoint, proceed with the joinpoint (or not) and execute advice after the joinpoint
  - The advice method must take a ***ProceedingJoinPoint*** parameter
  - The joinpoint is intercepted by the advice – allowing full control of execution before and after, and whether to execute the joinpoint
    - Calling ***proceed()*** on the *ProceedingJoinPoint* executes the joinpoint

```
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;
// Much detail omitted ...

  @Around("anyTeach()")
  public void helpTeach(ProceedingJoinPoint joinPoint)
                                                  throws Throwable {
    System.out.println("before: Turning Display Projector On");
    joinPoint.proceed(); // Execute the joinpoint
    System.out.println("after: Ahhh, Bringing teacher a drink");
  }
```

Around Example

- Calling *teach()* on a *JavaInstructor* instance will output:

  before: Turning Display Projector On

  Spring is Cool

  after: Ahhh, Bringing teacher a drink

- We don't necessarily have to call *proceed()*
  - For example, if we have logic in *helpTeach()* to check if the display projector is broken, we might not call *proceed()*

```
@Around("anyTeach()")
public void helpTeach(ProceedingJoinPoint joinPoint)
                                        throws Throwable {
  if (turnOnDisplayProjector()) {
    joinPoint.proceed();
    System.out.println("after: Ahhh, Bringing teacher a drink");
  } else {
    System.out.println("Class cancelled - broken projector");
  }
}
```

LEARNINGPATTERNS

Lab 9.2 – More Spring AOP

Lab 9.2 – More Spring AOP

Lab

- **Overview**: In this lab, we will work with some more Spring AOP capabilities
  - We'll use additional kinds of advice and additional designators
  - We will still keep it fairly simple – it's easy to lose the concepts in a sea of advice and pointcuts
- **Objectives**:
  - Use more of the Spring AOP capabilities
- **Builds on previous labs**: 9.1
- **Approximate Time**: 20-40 minutes

Using Around Advice

Lab

Tasks to Perform

- Continue to work in the **Lab09.1** project
- Open up the file *TeachingSupport.java*
  - Create an ***@Around*** advice on the *anyTeach()* pointcut you defined in the previous lab
  - Define it on the method *helpTeach()*
  - The advice should output something before the joinpoint, proceed with the joinpoint, and output something after the joinpoint
- Run the program - you should see output from both your advice and from the call to *teach()*
- Try commenting out the call to *proceed()* in the advice
  - Run your program – what do you see? Why?
  - Remove the comment so everything runs normally

[Optional] AfterReturning Advice

Lab

[Optional] Tasks to Perform - Depends on previous labs optional

- Continue to work in *TeachingSupport.java*
  - Modify the *inInfoSource()* pointcut expression to use the following to specify the execution of *getData()* only

```
execution(* com.javatunes.teach.InfoSource.getData(..))
```

- Modify the *@Before* advice that uses the *inInfoSource()* pointcut to be *@AfterReturning* advice instead
  - Modify the pointcut expression to access the return value of the joinpoint (see the notes page of the *@AfterReturning* example code)
  - Print out the return value in your advice method
- You can't modify the return value using *@AfterReturning*
  - If you wanted to do this, you'd have to use Around advice
  - With Around advice, you have a chance to get the return value from *proceed()*, and do something with it, or return a different value it you want

[Optional] Additional Things to Try

Lab

[Optional] Tasks to Perform

- Try using additional designators, and designator expressions
  - You may need to define new pointcuts for this – make up your own
- Try accessing arguments in your advice
  - You'll need to define a new method on *Teacher* that takes an argument, implement it in *JavaInstructor*, and call it in *TeachMeSpring*
  - You'll then need to write your advice to access the arguments (see the notes on Around advice in the earlier Module slides)

STOP

XML Based AOP Support

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

Defining Aspects Using XML

- Spring AOP supports defining aspects with XML in the Spring configuration files, which has some advantages
  - You don't need the source code of the Aspect class
  - Centrally organizes AOP configuration
- XML configuration uses tags in the Spring ***aop*** namespace
  - It supports the same capabilities as the @AspectJ style aspects
  - We'll cover the syntax of using the XML configuration here
- When you use the XML configuration, the aspects are completely normal Java classes
  - The pointcut and advice information is captured in the XML
  - There is no evidence of it in the Java class

AOP XML Configuration Example

- The fragment below shows an example aop XML configuration
  - **<aop:config>** is the top level element for almost all AOP configuration
  - *<aop:pointcut>* defines the *anyTeach* pointcut seen before
  - *<aop:aspect>* defines an aspect and specifies the aspect class as *TeachingSupport* (through the *teachSupport* ref)
  - *<aop:before>* defines *setupDisplayProjector* as before advice using the *anyTeach* pointcut
  - This aspect acts exactly like the one we defined with @AspectJ

```
<beans ... >
<bean id="teachSupport" class="com.javatunes.teach.TeachingSupport"/>

<aop:config>
   <aop:pointcut id="anyTeach" expression="execution(* teach(..))"/>
   <aop:aspect id="teachAspect" ref="teachSupport">
     <aop:before pointcut-ref="anyTeach"
                 method="setupDisplayProjector"/>
   </aop:aspect>
</aop:config>
</beans>
```

Specifying Advice With XML

- The same advice types we've seen earlier are supported
  - *aop:before, aop:after, aop:after-returning, aop:after-throwing, aop:around*
  - They act the same as the @AspectJ counterparts we've seen
  - For example, for *aop:around*, the first parameter of the advice method has to be of type *ProceedingJoinPoint*
  - e.g., if *TeachingSupport* defined the following method

```
public void helpTeach(ProceedingJoinPoint joinPoint)
                                    throws Throwable {
```

  - We could specify *helpTeach()* as *around* advice as shown below:

```
<aop:config>
  <aop:pointcut id="anyTeach" expression="execution(* teach(..))"/>
  <aop:aspect id="teachAspect" ref="teachSupport">
    <aop:around pointcut-ref="anyTeach" method="helpTeach"/>
  </aop:aspect>
</aop:config>
```

LEARNINGPATTERNS

Lab 9.3 – Using AOP XML Configuration [Optional]

[Optional] Lab 9.3 – XML Configuration

Lab

- **Overview**: In this (optional) lab, we will use the XML configuration to define our aspects
  - We'll define the same aspects and advice as in the previous lab
- **Objectives**:
  - Understand and use the Spring AOP XML configuration
- **Builds on previous labs**: 9.2
- **Approximate Time**: 20-30 minutes

Using Before Advice

Lab

Tasks to Perform

- Continue to work in the **Lab09.1** project
- Copy the Lab09.1 project to another project if you want to save a copy of the annotation based lab
- **Open TeachingSupport.java** for editing
  - **Comment out or remove all the pointcuts and annotations**
  - You can comment out the imports if you want, but you'll need the import of *ProceedingJoinPoint*
- **Open *applicationContext.xml*** for editing
  - Add in configuration elements to define the *anyTeach* pointcut, an aspect using a *TeachingSupport* bean, and *before* advice with *setupDisplayProjector* as the advice method
  - See the Module slides, and your @AspectJ version for help
- **Run the program**, and you should still see output from your before advice, and from the call to *teach()*

Using Other Kinds of Advice

Lab

Tasks to Perform

- Continue to work in *applicationContext.xml*
  - Add the configuration to define *around* advice in your aspect with *helpTeach* as the advice method
  - The configuration follows the same structure as before advice
- **Run the program**, and you should see output from your before advice, your around advice, and from the call to *teach()*
- **Optional**: Add the configuration to define after-returning advice in your aspect, with *checkData* as the advice method
  - You'll need to include a returning attribute in the advice, as well as the *pointcut-ref* and *method* attributes

  returning="retVal"
  - **Run the program** and see that the output is what you expect
- You can see that the XML configuration isn't hard to use
  - It's different syntax, but the exact same structure

STOP

Other Considerations

Overview

Spring AOP Introduction

Pointcut Expressions and Advice

XML Based AOP Support

Other Considerations

Spring Proxies and Direct Invocation

- Spring AOP works by using proxies that intercept a method invocation and then add in the AOP functionality
  - We've seen this with Spring's transaction support also
  - It has similar results here
- You must access an object through the container for Spring AOP to work
- If you are in a class, and call another method on the same class, **the invocation won't be advised**
  - The invocation is via the *this* reference, not through a proxy
- As with transactions, you can configure load time weaving

```
<context:load-time-weaver aspectj-weaving="on"/>
```

  - See the transaction Module for more details

More on Spring Proxies

- In general, your Spring managed beans should be referenced through interface types, not through concrete types
  - This is the standard and recommended practice
  - When you do this, the AOP proxies are created via Java dynamic proxies

- If your class doesn't implement any interfaces (bad practice), you can still use AOP
  - Spring will proxy the class using CGLIB – which creates subclasses for the proxy type, but this has some ramifications
  - You can't advise final methods
  - You'll need the cglib jars in your classpath
  - Your constructor will be called twice (once for the bean instance, once for the proxy subclass instance)

Issues with AOP

- We've seen some of the capabilities of AOP
  - How it complements OO programming by providing a way to inject cross-cutting concerns into your code
- Let's now look at some of AOP's issues
- **You can't see what's going on** just by looking at your business code
  - An aspect might be injected that you don't know anything about
  - This makes the code harder to understand and maintain
  - Tools also don't have much support for visualizing the crosscutting concerns
- AOP is powerful – a **mistake** may have large consequences
  - For example, if a programmer makes a mistake in defining a pointcut, advice may be applied unintentionally

Issues with AOP

- It's easy to **unintentionally and unknowingly** change the way AOP applies to your code
  - e.g., if a programmer simply renames a method or module, it can affect the way AOP applies to that method or module
- It's **hard to debug** or find out the cause of failures
  - If something goes wrong in a system, it may be difficult to pinpoint if it's happening in core business code, in advice code, if a complex pointcut expression is not exactly right, if somebody changed a name so advice is being improperly applied, and etc.
  - This also means it's hard to assign responsibility (and even legal liability) for a bug or failure
- There are **security issues**
  - What happens if someone maliciously injects around advice into your security verification code so it allows them in?

Is AOP Worth It ?

- Like with any powerful capability, it needs to be used appropriately
  - It should only be used where it is a true cross-cutting concern AND it makes your life easier
  - Overuse or overly complex use can be problematic as it can make your code harder to write, understand, and maintain
  - Instead of making your life easier – which was the whole reason we would even consider it
- Spring transactions are a nice use of AOP
  - It makes TX behavior much easier to understand and control
- When used appropriately, it can be a very useful technique
  - But it is powerful, and should be used with care – even caution

Other AOP Capabilities and Functionality

- There is an enormous amount of capability in Spring AOP
  - We've given you enough to understand the mechanisms and the structure - but there is much more detail

- Spring AOP also gives you the capability to use more extensive capabilities
  - The Spring implementation is designed for ease of use, and usability in most common circumstances
  - For requirements that require more, you can use more capabilities, such as load-time-weaving

  - The Spring and AspectJ docs have a large amount of information for you to use to dig in deeper

Review Questions

- What is AOP, and what kinds of issues does AOP address?
- What is an aspect, a pointcut, advice?
- What are @AspectJ annotations, and how do they relate to Spring AOP
- How does Spring weave AOP advice into your program?
- What is a pointcut expression, and how do you define one with @AspectJ and Spring AOP
- How do you define advice with @AspectJ and Spring AOP
- How do you use AOP with Spring 2.x schema support?
- What are some of the issues surrounding AOP

Lesson Summary

- Dealing with crosscutting concerns (functionality) that extends across multiple modules is difficult in large software systems.
  - AOP allows you to encapsulate crosscutting concerns in one place called aspects
- **Aspects** can define the following (among other things)
  - **Advice**: The action taken by an aspect
  - **Joinpoint**: Point in a program where advice may run
  - **Pointcut**: A specification that matches joinpoints
- Spring uses **@AspectJ**'s annotations and compiler to define AOP types - while keeping its own proxy-based runtime
  - Annotations like *@Aspect*, *@Pointcut* and *@Before*, let you fully define the aspect in a Java class
  - Aspect code runs via Spring's proxy mechanism

Lesson Summary

- **Pointcuts** use pointcut expressions in the AspectJ language to specify what joinpoints are of interest
- **Advice** is associated with a method via one of the advice annotations, such as ***@Before***
  - The method body implements the advice
  - All the AOP capabilities can be also be configured with Spring XML elements from the ***aop*** namespace
- While very powerful, AOP raises some significant issues
  - These arise from the same characteristic that makes it powerful
  - You are invisibly injecting advice into your existing code base
  - There are many usability, maintenance, and security issues that can arise from this

LEARNINGPATTERNS

Recap

Recap of what we've done

- We've covered a large part of the Spring Framework
  - The **core Spring container** with bean configuration, Dependency Injection, resource access, bean lifecycle management, events, etc.
  - **Database access** with Spring, including the use of templates such as *JdbcTemplate* and Spring's support for building DAOs based on JDBC, Hibernate, and JPA
  - **Controlling transactions** declaratively with Spring declaratively using Spring annotations and XML configuration
  - Integrating Spring with **Web apps** using *ContextLoaderListener*
  - How to use **Spring security** to add access protection to your apps
  - Creating **RESTful services** using Spring MVC
  - The powerful **AOP** features of Spring, and Spring's @AspectJ and XML configuration

What Else is There

- Spring is a very large framework
  - We covered the structure and core ideas of all the areas we touched on
  - There is lots of opportunity for you to learn as you get more familiar with Spring
  - Read the documentation, read the books, use the Web
  - There is a lot of information out there to help you

- There are also additional capabilities that Spring provides
  - Look at the Spring reference manual - it has a lot of information on what the Spring capabilities are
  - There is much to explore in the wide Spring world

Resources

- The **Spring documentation**
  - The reference manual, API docs, and samples
  - They're all available in the full Spring download
- **http://www.springsource.org**
  - The home of the Spring Framework
- **Spring in Action** – Third Edition by Craig Walls
  - An excellent book, full of useful examples and explanations
  - It's big - 700 pages +
- There are many more - these are a great place to start