



The Spring Framework Part 1: Fundamentals, Injection, AOP, Beginning MVC

VERHOEF TRAINING

103 Hillside Avenue

West Caldwell, NJ 07006

1-800-631-0410

www.verhoef-training.com

Spring Framework(Part 1)

Fundamentals, Injection, AOP, Beginning MVC

Spring Framework Table of Contents

I.	Java EE and Spring	3
II.	Spring IOC	27
III.	Spring JDBC	79
IV.	DAOs, ORM Frameworks, and Spring	109
V.	Spring MVC	143
VI.	MVC Controllers: Using Annotations	171
VII.	Bindings and Tags	189
VIII.	AspectJ and AOP	199
IX.	Spring AOP	227
X.	JUnit	249

2

©2015 Computer Trilogy, Inc. ALL RIGHTS RESERVED

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the Author—Computer Trilogy, Inc. All other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs. THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

I. Java EE and Spring

3

Problems with J2EE

- ❖ J2EE was overly complex for most needs
 - Distributed objects not always needed
 - JNDI programming overhead unnecessary
 - Excessive try/catch handling
 - Difficult to unit test
 - Special environments needed: EJB Container
 - New implementations, such as *Ruby on Rails*, are providing faster time-to-production

4

Notes:

The Need for Something Better

- ❖ J2EE provides low-level enterprise services
 - Distributed services
 - Object naming
 - Transaction control, etc.
- ❖ Something is needed to manage an entire enterprise architecture

Enter the ***Spring*** Framework!

5

Notes:

While the J2EE is one of the most comprehensive enterprise service APIs, it requires lots of repetitive, unnecessary, and low-level code writing to achieve its goals. It also requires heavyweight containers and often forces us to implement features we do not need. This has opened the door for J2EE critics to attack its performance over the years

What is Spring?

❖ A Lightweight Framework

- Provides services for all tiers of an application
- Uses POJOs without heavy requirements about the “rules” for those objects (“lightweight”)
- Primary services include:
 - Inversion of Control (IOC)
 - JDBC Abstraction Layer
 - AOP
 - Spring MVC

6

Notes:

What is a POJO? A POJO is a term that was originally used to describe back-end components that did not require an EJB Container.

Do I Really Need Another Framework?

- ❖ It emphasizes development of business objects
 - You spend less time on JEE APIs
- ❖ Only need to incorporate the services *you* desire
- ❖ Testing and Integration are made simpler



7

Notes:

State of Spring

❖ Rod Johnson

- Founding Creator of Spring
- Author of *J2EE Development w/o EJB*

❖ Version History and More see:

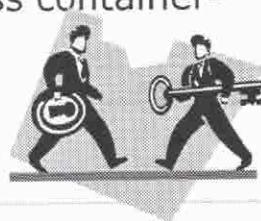
- http://en.wikipedia.org/wiki/Spring_Framework

8

Notes:

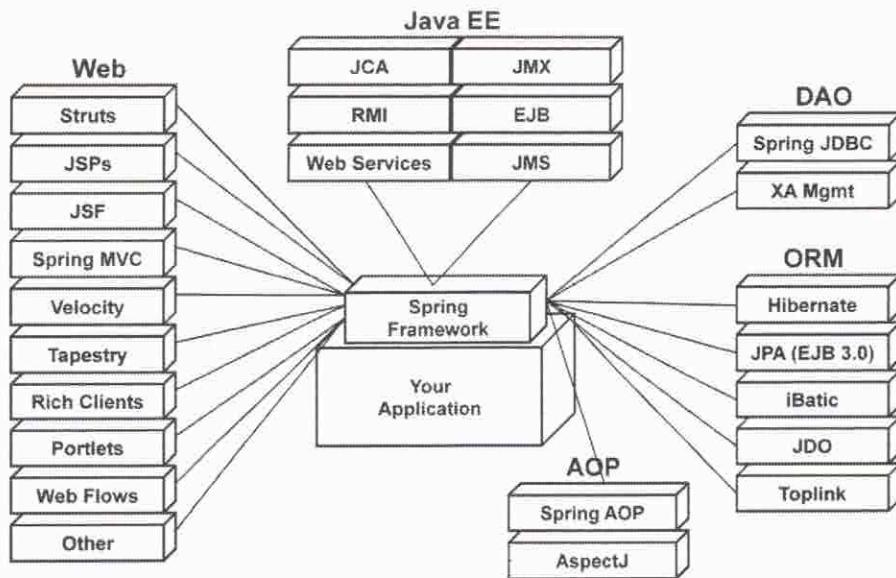
Compatibilities

- ❖ Spring 2.0 works with Java 1.3 or later
- ❖ Spring 2.5 requires Java 1.4 or later
- ❖ Spring 3.0 requires Java 1.5 or later
- ❖ Spring 4.0 requires Java 1.8 or later
- ❖ Spring does not require Java EE libraries or a Java EE container
 - Unless specifically trying to access container-based services



Notes:

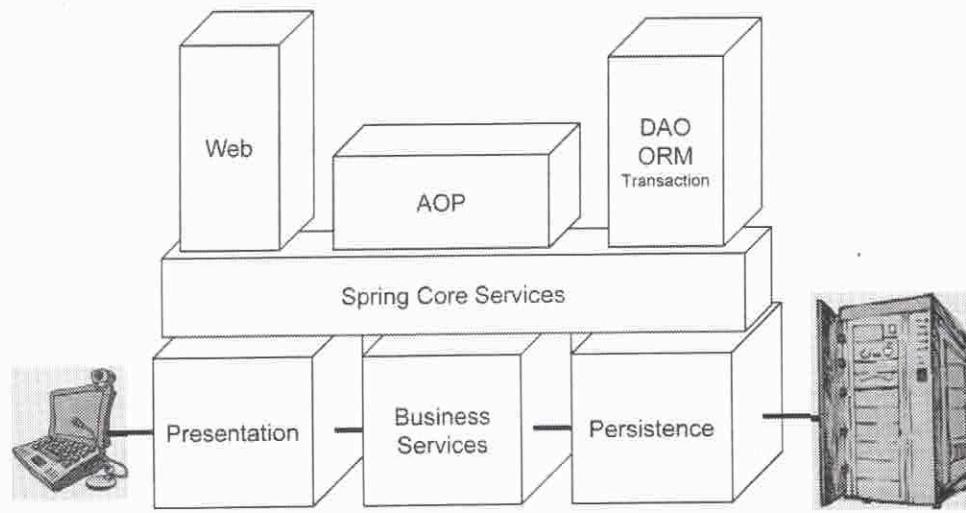
Spring and Integration



10

Notes:

Spring Framework



11

Notes:

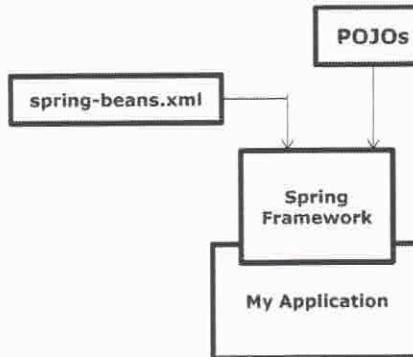
IOC Introduction

❖ What is IOC?

❑ Inversion of Control

- Lets you externalize and manage dependencies
- Improves architecture
- Adds flexibility to create and change services as needed

Spring's IOC Container provides the capabilities to inject beans automatically into your applications without using a Service Locator to ask for it



12

Notes:

An Initial Application

- ❖ Let's begin with a simple example...
- This application returns the day of the week for a given year, month, day

```
public class Driver
{
    public static void main(String[] args)
    {
        int year  = 2006;
        int month = 9;                      // October
        int day   = 16;
        Calendar c = new GregorianCalendar(year, month, day);
        System.out.println(
            new SimpleDateFormat("EEEE").format(c.getTime()));
    }
}
```

This application lacks a separation of its services and presentation layer

13

Notes:

Outputs 'Monday'

Imports have been omitted.

The Initial App - Improved

```
public class Driver {  
    public static void main(String[] args) {  
        int year = 2006;  
        int month = 9; // October  
        int day = 16;  
  
        String dayOfWeek = DayOfWeekService.getDayOfWeek(  
                                         year, month, day);  
        System.out.println(dayOfWeek);  
    }  
}  
  
public class DayOfWeekService {  
    public static String getDayOfWeek(int year, int month, int day)  
    {  
        Calendar c = new GregorianCalendar(year, month, day);  
        return new SimpleDateFormat("EEEE").format(c.getTime());  
    }  
}
```

While improved, what little there is of the UI layer (the driver) is *too highly coupled* with the service.

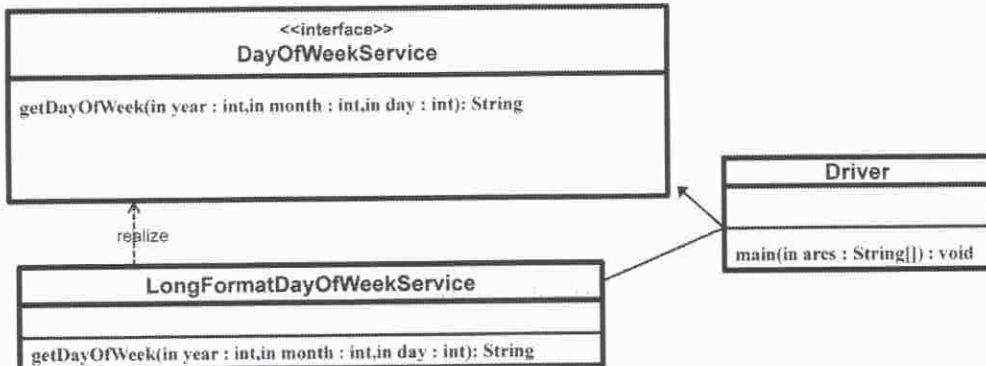
14

Notes:

This improved version still suffers several problems. The presentation layer (the Driver class, in this case) is too highly coupled with the service, therefore, changes made to the service layer might cause compile or runtime errors to develop in the presentation layer.

Programming to Interfaces

```
Public interface DayOfWeekService
{
    public String getDayOfWeek(int year, int month, int day);
}
```



15

Notes:

The first step towards creating a decoupled layered solution is to abstract away the concrete classes, basing them instead on interfaces. In this scenario, a new concrete class, called `LongFormatDayOfWeekService`, is created based on the interface, `DayOfWeekService`.

Programming to Interfaces ...

```
public class Driver {  
    public static void main(String[] args) {  
        int year = 2006;  
        int month = 9; // October  
        int day = 16;  
        DayOfWeekService service = new LongFormatDayOfWeekService();  
        String dayOfWeek = service.getDayOfWeek(year, Month, Day);  
        System.out.println(dayOfWeek);  
    }  
}
```

App is now dependent
on two components

```
public class LongFormatDayOfWeekService implements  
DayOfWeekService {  
    public String getDayOfWeek(int year, int month, int day) {  
        Calendar c = new GregorianCalendar(year, month, day);  
        return new SimpleDateFormat("EEE").format(c.getTime());  
    }  
}
```

16

Notes:

This solution uses interfaces without any special benefit. Now the driver is dependent upon both the interface and the implementation class, LongFormatDayOfWeekService.

The Service Locator Pattern

```
public class Driver {  
    public static void main(String[] args) {  
        int year = 2006;  
        int month = 9;// October  
        int day = 16;  
        DayOfWeekService service =  
            (DayOfWeekService) Locator.getService("long");  
  
        String dayOfWeek = service.getDayOfWeek(year, month, day);  
        System.out.println(dayOfWeek);  
    }  
}  
  
public class Locator {  
    public static Object getService(String serviceName) {  
        Object service = null;  
        // instantiate service based on serviceName  
        return service;  
    }  
}
```

17

Notes:

The service locator pattern provides a nice means for not coupling the implementation with the presentation. In this example, a class called "Locator" is used to obtain and instantiate the resource associated with the string literal "long". In other words, when Locator's getService() method is passed the value "long", an instantiated service is returned.

Writing Your Framework

- ❖ Using a Locator will decouple your application's layers
 - It takes time to build and test a flexible framework that decouples components
 - This solution is a classic approach toward dependency removal
 - Sometimes called a "pull" or dependency lookup



18

Notes:

Using Spring simply for its Service Locating capability is like using the Space Shuttle to travel from Denver to Salt Lake City.

Spring JARs

org.springframework.aop
org.springframework.asm
org.springframework.aspects
org.springframework.beans
org.springframework.context.support
org.springframework.context
org.springframework.core
org.springframework.expression
org.springframework.instrument.tomcat
org.springframework.instrument
org.springframework.jdbc

org.springframework.jms
org.springframework.orm
org.springframework.oxm
org.springframework.spring-library
org.springframework.test
org.springframework.transaction
org.springframework.web.portlet
org.springframework.web.servlet
org.springframework.web.struts
org.springframework.web



19

Notes:

Building a Spring App



Create a Java Project
(assumes an Eclipse-based IDE)



Add (to the build path) the Jakarta Commons JARs



Add JARs (to the build path)

20

Notes:

Create a XML Config File



```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
    <bean id="long" class="com.company.springclass.examples.ch01.  
                           service.LongFormatDayOfWeekService"></bean>  
  
    <bean id="short" class="com.company.springclass.examples.ch01.  
                           service.ShortFormatDayOfWeekService"></bean>  
  
  </beans>
```

21

Notes:

In this example, the long class names have been wrapped onto the next line. In practice, they should be continuous lines.

be an 3 STANDARD
1. GETTER / SETTER X
2. IMPLEMENT SERIALIZABLE
3. DEFAULT CON STRUCTOR



Create the Main App

```
public class Driver {  
    public static void main(String[] args) {  
  
        int year = 2006, month = 9, day = 16;  
  
        ApplicationContext context = new  
            ClassPathXmlApplicationContext("beans.xml");  
  
        DayOfWeekService service = (DayOfWeekService)  
            context.getBean("long");  
  
        String dayOfWeek = service.getDayOfWeek(year, month, day);  
        System.out.println("Long version: " + dayOfWeek);  
  
        service = (DayOfWeekService) context.getBean("short");  
        dayOfWeek = service.getDayOfWeek(year, month, day);  
        System.out.println("Short version: " + dayOfWeek);  
    }  
}
```

22

Notes:

The following imports were omitted from the code above:

```
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import com.company.springclass.examples.ch01.service.DayOfWeekService;
```

Spring Capabilities

- ❖ This example only begins to scratch the surface!
 - Note how easy it is to change out services
 - Spring's IOC capabilities go well beyond acting as a Service Locator!



23

Notes:

Summary

- ❖ Spring is the *most powerful IOC container available*
 - It provides and handles many of the lower-level tasks that make Java/J2EE development tedious
 - Writing Spring-enabled apps is a matter of wiring up beans in an XML configuration file
 - Use a bean factory (context) to obtain those beans in your applications

24

Notes:

Lab 1

❖ *Creating Multiple Services*

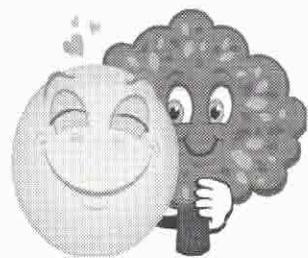
- Follow the instructions found in the ***SpringLab01/instructions.html*** file to complete the exercise



25

Notes:

This page intentionally left blank!



26

26

II. Spring IOC

27

Spring IOC Overview

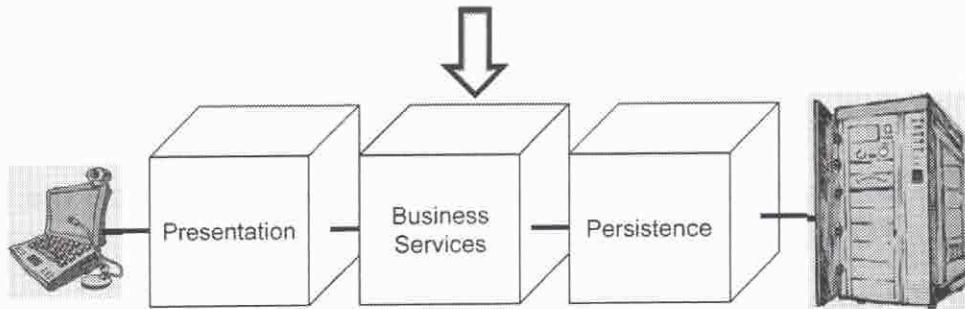
- ❖ IOC and Types of DI
- ❖ Spring Definition Files
- ❖ Creating Factories and Services
- ❖ Bean Scopes
- ❖ Other Bean Properties



28

Notes:

Inversion of Control



Spring IOC uses a special form of control called:

Dependency Injection

29

Notes:

Types of Dependency Injection

- ❖ Injection uses the container to insert objects into your application automatically
 - Locators use a lookup technique to "pull" objects into the application
- ❖ There are 3 types of *Dependency Injection*
 - Setter Injection
 - Constructor Injection
 - Method Injection



30

Notes:

Setter Injection

applicationContext-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance  
        xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
    <bean id="customerBean"  
          class="com.company.springclass.examples.ch02.Customer">  
        <property name="name">  
            <value>John Smith</value>  
        </property>  
        <property name="customerId">  
            <value>1</value>  
        </property>  
    </bean>  
  
</beans>
```

These values are
injected into the bean
by the IOC Container

31

Notes:

Setter Injection ...

```
public class Customer
{
    private String name;
    private int customerId;

    public Customer() {}
    public Customer(String name, int customerId) {
        this.name = name;
        this.customerId = customerId;
    }

    public String getName() { return name; }
    public int getCustomerId() { return customerId; }

    public void setName(String name) { this.name = name; }
    public void setCustomerId(int id) { customerId = id; }

    public String toString() { return name + " " + customerId; }
}
```

This customer bean,
controlled by Spring, is a
simple POJO

32

Notes:

Setter Injection ...



```
public class Driver
{
    public static void main (String[] args)
    {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "applicationContext-beans.xml");

        Customer customer = (Customer) ctx.getBean("customerBean");

        System.out.println(customer);  Outputs: John Smith 1
    }
}
```

33

Notes:

Constructor Injection

applicationContext-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="  
           http://www.springframework.org/schema/beans  
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
    <bean id="customerBean"  
          class="com.company.springclass.examples.ch02.Customer">  
        <constructor-arg>  
            <value>Andrew MacDonald</value>  
        </constructor-arg>  
        <constructor-arg>  
            <value>2</value>  
        </constructor-arg>  
    </bean>  
  
</beans>
```

These values are
injected into the bean
by the IOC Container via
a Constructor

34

Notes:

Constructor Injection ...



```
public class Driver
{
    public static void main (String[] args)
    {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "applicationContext-beans.xml");

        Customer customer = (Customer) ctx.getBean("customerBean");

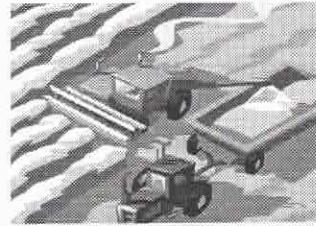
        System.out.println(customer); Outputs: Andrew McDonald 2
    }
}
```

35

Notes:

Injecting Beans into Beans

```
public class Order {  
    private int orderId;  
    private double totalPrice;  
    private Customer customer;  
  
    public int getOrderId() { return orderId; }  
    public double getTotalPrice() { return totalPrice; }  
    public Customer getCustomer() { return customer; }  
  
    public void setOrderId(int orderId) { this.orderId = orderId; }  
    public void setTotalPrice(double totalPrice) {  
        this.totalPrice = totalPrice; }  
    public void setCustomer(Customer customer) {  
        this.customer = customer; }  
  
    public String toString() {return orderId + " " +  
        customer.toString() + " " + totalPrice; }  
}
```



36

Notes:

Injecting Beans into Beans ...

applicationContext-beans.xml

```
<bean id="orderBean"
      class="com.company.springclass.examples.ch02.Order">
    <property name="orderId">
      <value>100</value>
    </property>
    <property name=""customer">
      <ref local="customerBean"/>
    </property>
  </bean>

  <bean id="customerBean" <-->
    class="com.company.springclass.examples.ch02.Customer">
    <property name="name">
      <value>John Smith</value>
    </property>
    <property name="customerId">
      <value>1</value>
    </property>
  </bean>
```

Outputs: 100 John Smith 1 0.0

37

Notes:

For beans defined within other beans (in Spring these are called inner beans), the singleton, params, and name attributes are effectively ignored for that inner bean.

Injecting Beans into Beans ...

```
public class Driver
{
    public static void main (String[] args)
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(
                "applicationContext-beans.xml");
        Order order = (Order) context.getBean("orderBean");
        System.out.println(order);
    }
}
```

Outputs: 100 John Smith 1 0.0



38

Notes:

XML Configuration Files

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

<!-- bean definitions here -->

</beans>
```



39

Notes:

Understanding the Schema Attributes

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

    <!-- bean definitions here -->

</beans>
```

Sets the default namespace as the one containing the Spring XML tags.

Identifies the namespace used for the Spring tags.

40

Notes:

Spring Multiple Schemas

Many of the Spring config tags have been placed into different schema

```
<?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:util="http://www.springframework.org/schema/util"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jee="http://www.springframework.org/schema/jee"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-3.0.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/spring-jee-3.0.xsd"
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context3.0.xsd>
```

41

Notes:

The Spring config file has expanded in terms of the tags that may be used. So, it now exists in multiple schema. If you use tags from a specific schema, for example, the context schema, then you must include the *xmlns:* prefix for that schema, and in the *xsi:schemaLocation* attribute the *xsd* file and its url must be included.

In the example above, namespace values are only needed if tags from that namespace are used.

The <bean> Element

```
<bean id="MyBean"
      class="org.company.app.beans.MyBean"
      name="CustomerBean, Customer, CustBean"
      scope="singleton | prototype | request | session"
      autowire="true | false"
    >
    <property />
    <constructor-arg />
</bean>
```

The most important element
within a Spring config file is
the <bean> element.

42

Notes:

These attributes are explained in more detail on the following slides.

***id* and *name* Attributes**

```
<bean id="myBean" class="java.lang.String"
      name="myOtherBean,anotherBean">
    <constructor-arg>
      <value>It's Spring Time!</value>
    </constructor-arg>
</bean>
```



```
ApplicationContext context = new
ClassPathXmlApplication (
    "applicationContext-beans.xml");
String myBean = (String) context.getBean("myBean");
String myOtherBean = (String) context.getBean("myOtherBean");
String anotherBean = (String) context.getBean("anotherBean");

System.out.println(myBean);
System.out.println(myOtherBean);
System.out.println(anotherBean);
```

Outputs 'It's Spring
Time' 3 times.



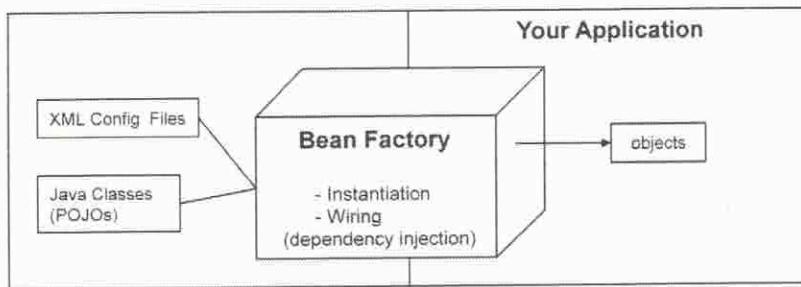
43

Notes:

The name attribute of the bean element serves as an alias for that bean. You may use the id or the name element to obtain a bean.

The Spring BeanFactory

- ❖ The Spring IOC container is known as a “Bean Factory”



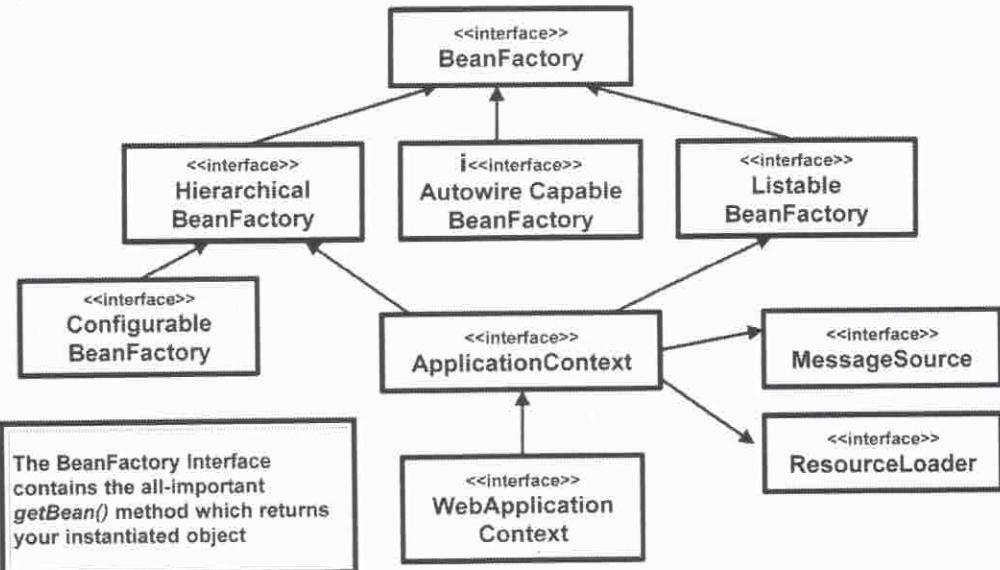
- ❖ Each BeanFactory contains and manages the beans declared in its configuration files.

44

Notes:

There is also an API in the BeanFactory interface for dynamically establishing beans and dependencies, the use of XML is by far the most common approach.

BeanFactory Hierarchy



45

Notes:

Spring Config File Naming Conventions

- ❖ Spring **config** files are often broken down into functional types

- Examples:

- applicationContext-beans.xml
 - applicationContext-dao.xml
 - applicationContext-services.xml



46

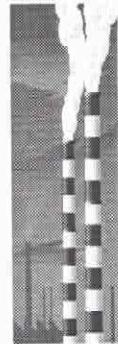
Notes:

While the names are not very important, a larger Spring-based application should utilize a naming scheme that will make it easier to manually locate your bean definitions. This example identifies the factory and the type of beans the factory will be using.

The BeanFactory Interface

- ❖ All *BeanFactory* Classes implement this interface
`org.springframework.beans.factory.BeanFactory`
- ❖ The *BeanFactory* Interface methods

```
Object getBean(String name)
Object getBean(String name, Class type)
Class getType(String name)
boolean containsBean(String name)
boolean isSingleton(String name)
String[] getAliases(String name)
```



47

Notes:

The ApplicationContext

- ❖ The *ApplicationContext* is the most commonly used factory within Spring
 - It *is* a *BeanFactory*
`org.springframework.context.ApplicationContext`
 - It contains additional capabilities over the *BeanFactory*, such as
 - Internationalization support
 - Ability to load message resources
 - Ability to create parent/child hierarchies so an entire web application or just a servlet can be aware of a factory

48

Notes:

Creating the Container

❖ Examples of Creating the IOC Container

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("applicationContext-beans.xml");
```

Locates a bean factory definition file on the classpath

```
ApplicationContext context = new FileSystemXmlApplicationContext(  
"/apps/config/applicationContext-beans.xml");
```

Locates a bean factory definition file from a file system path



49

Notes:

Generally the ClassPathXmlApplicationContext is the chosen concrete beanfactory implementation.

Creating the Container ...

```
String[] definitionFiles = {"applicationContext-dao.xml",
                            "applicationContext-services.xml"};
ApplicationContext context = new
    ClassPathXmlApplicationContext(definitionFiles);
```

→ Configures beans from multiple files

```
<beans ...>
    <bean id="CustomerDataSource"
          class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/jdbc/CustomerDS"/>
    </bean>
</beans>
```

applicationContext-dao.xml

```
<beans ...>
    <bean id="GetCatalogService"
          class="com.company.app.services.GetCatalog">
    </bean>
</beans>
```

applicationContext-services.xml

50

Notes:

Spring-based Services

- 1) Create the Service Interface
(CatalogManager)
- 2) Create a Service Implementation
(CatalogManagerImplementation)
- 3) "Wire up" the Beans
- 4) Create an IOC Container
- 5) Get and work with the Beans



51

Notes:



Create the Service Interface

```
package com.company.springclass.services;  
  
import java.util.List;  
import com.company.springclass.beans.Product;
```

```
public interface CatalogManager  
{  
    public List getProducts() ;  
    public Product getProduct(int productId) ;  
}
```

52

Notes:

Create a Service Implementation



```
public class CatalogManagerImplementation implements CatalogManager
{
    List products;

    public CatalogManagerImplementation(List products) {
        this.products = products;
    }

    public Product getProduct (int productId)
    {
        return null;
    }

    public List getProducts()
    {
        return products;
    }

    public String toString() {
        return (products == null ? "" : products.toString());
    }
}
```

53

Notes:



Wiring Up Services

```
<beans ...>
```

```
  <bean id="catalog"
        class="com.company.springclass.services.CatalogManagerImplementation">
    <constructor-arg>
        <list>
            <ref local="productBean"/>
        </list>
    </constructor-arg>
  </bean>

  <bean id="productBean" class="com.company.springclass.beans.Product">
    <constructor-arg><value>1</value></constructor-arg>
    <constructor-arg><value>Socks</value></constructor-arg>
    <constructor-arg><value>1.89</value></constructor-arg>
    <constructor-arg><value>4.99</value></constructor-arg>
    <constructor-arg><value>5</value></constructor-arg>
  </bean>
</beans>
```

Passes a list of product beans (only one) into the constructor of the CatalogManagerImplementation class

54

Notes:



Creating an IOC Container

```
package com.company.springclass.examples;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.company.springclass.services.CatalogManager;

public class Driver {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(
                "applicationContext-beans.xml");

        // obtain and work with beans...
    }
}
```

55

Notes:

5

Work with the Beans

```
CatalogManager catalog =  
    (CatalogManager) context.getBean("catalog");  
  
System.out.println(catalog);
```



56

Notes:

Lab 2: Using Spring's DI

- ❖ In this exercise, you will create a Catalog and use Spring's constructor injection to insert products
- ❖ Open and read ***instructions.html*** in the **SpringLab02** project and follow the additional steps provided



57

Notes:

Beans Scopes

- ❖ Spring Beans can be scoped several different ways

- Singleton (*default in all versions of Spring*)

- Prototype

- Request

- Session

Spring-aware web
applications only

```
<bean id="myService" class="com.company.app.services.MyService"  
scope="prototype"/>
```

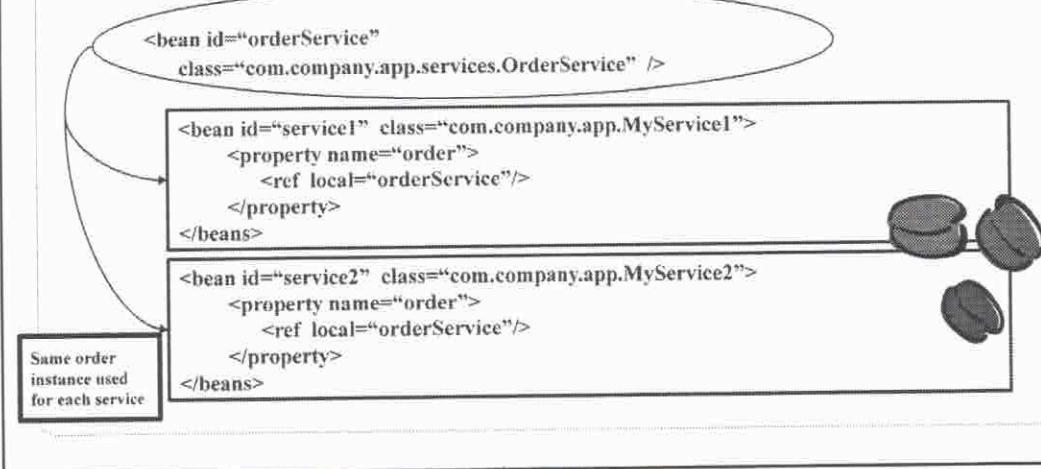
58

Notes:

The singleton type is still the default. Not mentioning a scope will default to scope="singleton". The custom scope allows you to define a scope that is relevant to your application. To create an object that serves as a scope object, it must implement the Spring Scope interface and then register itself with the BeanFactory (ApplicationContext) using the registerScope() method.

Singleton Beans

- ❖ In Spring, only one instance of a Singleton bean will be created per Factory
 - In most design pattern strategies, Singletons exist on a per *ClassLoader* basis



Notes:

Spring-defined singleton beans are not Gang of Four-defined Singletons. It is entirely possible for a single spring configuration file to define multiple beans with the same class.

Prototype Beans

- ❖ A new instance is created for each **prototype** scoped bean whenever `getBean()` or a reference from another bean calls for it.

```
<bean id="orderService" scope="prototype"
      class="com.company.app.services.OrderService" />

<bean id="service1" class="com.company.app.MyService1">
    <property name="order">
        <ref local="orderService"/>
    </property>
</beans>

<bean id="service2" class="com.company.app.MyService2">
    <property name="order">
        <ref local="orderService"/>
    </property>
</beans>
```

instance1 →

instance2 →



60

Notes:

Use prototype scoped beans when the beans have stateful (instance field) values that should not be shared. After handing the prototype bean to the client, the IOC Container no longer manages that bean.

Request and Session Beans

- ❖ Beans defined with request scope will be instantiated for each HTTP request

```
<bean id="orderBean"  
      class="com.company.app.beans.OrderBean"  
      scope="request"/>
```

- ❖ Spring creates and properly scopes a new bean instance for each new HTTP Session created

```
<bean id="orderBean"  
      class="com.company.app.beans.OrderBean"  
      scope="session"/>
```

61

Notes:

Steps Toward Making Web Apps Spring-Enabled

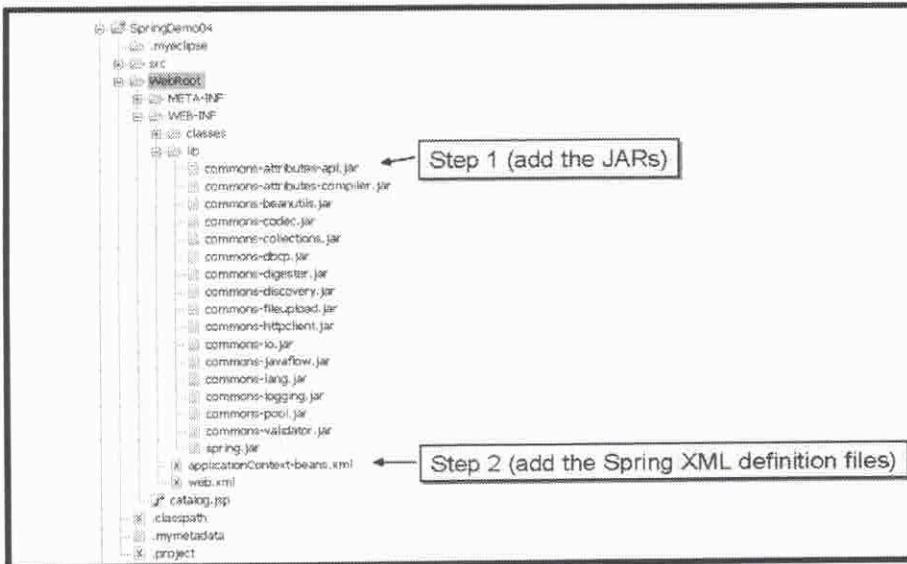
- ❖ Steps to make a web-app aware of Spring:
 1. Add the **Spring JARs** to the *WEB-INF/lib* directory
 2. Add the **Spring XML definition** files to the app (*usually WEB-INF directory*)
 3. Set a Context Parameter in *web.xml*, specifying the location of the **Spring XML definition** files
 4. Load the ApplicationContext with a **Listener** or **Servlet**
 5. Set up another **Listener** or **Filter** to allow beans to be scoped at the 'request' or 'session' level (*optional*)

62

Notes:

Step 5 is only required if you will be using beans scoped at the request or session level.

Spring-WebApp Configuration



63

Notes:

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

Spring-WebApp Configuration ...

```
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>
</web-app>
```

Step 3 (set a context param locating XML files)

Step 4 (set a listener to load the ApplicationContext)

Step 5 (set a listener for bean scoping)



Notes:

These two approaches are to be used for Servlet Containers that predate Servlet 2.4, or cannot otherwise make use of listeners:

Step 4 (alternate technique)

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

Step 5 (alternate technique)

```
<filter>
  <filter-name>requestContextFilter</filter-name>
  <filter-class>
    org.springframework.web.filter.RequestContextFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>requestContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Accessing Beans in a Web App

- ❖ Obtain the *ApplicationContext* from within your Web apps by using the Spring-provided utility **WebApplicationContextUtils**

```
ApplicationContext context =  
    WebApplicationContextUtils.getWebApplicationContext(  
        getServletContext());  
  
CatalogManager catalog =  
    (CatalogManager) context.getBean("catalog");  
  
out.println(catalog);
```

65

Notes:

This works from within Struts, JSF, Servlets, etc.

Session & Request Scopes

- ❖ Request scoped beans are instantiated on a per web-request basis
- ❖ Session scoped beans are instantiated on a per web-session basis

```
<bean id="catalog"
      class="com.company.springclass.services.CatalogManagerImplementation"
      scope="session">
    <constructor-arg>
      <list><ref local="productBean"/></list>
    </constructor-arg>
</bean>

<%= WebApplicationContextUtils.getWebApplicationContext(
    application).getBean("catalogBean") %>
```

This will come from the user's session object

66

Notes:

Spring takes care of the injecting of beans into the session or request objects. This example injects the Catalog bean into each session. To obtain the bean (say, from a JSP), use the code shown. The application object mentioned in the code above is the ServletContext object defined in any JSP. The example uses an expression tag to execute the Catalog bean's `toString()` method.

Lab 2b: Web-Enabling Spring

- ❖ In this exercise, you will *web-enable a spring application*
- ❖ This exercise uses the DayOfWeek service and the Catalog service to explore how to access various scoped beans
- ❖ Web framework integration (Struts, JSF, Spring MVC, etc.) will not be considered
- ❖ Read **instructions.html** in the *SpringLab02b* project and follow the additional steps provided

67

Notes:

Autowiring



- ❖ *Autowiring* within Spring is a means for the container to try to figure out dependencies automatically
- ❖ While this feature is powerful it should generally not be used for larger projects
 - It leaves the guess work up to the container
- ❖ *Autowiring* must be specified for each bean, therefore, beans can be excluded

68

Notes:

Autowiring for smaller projects might be okay, but larger projects might lead Spring to guess incorrectly on which beans should be wired together.

Autowiring through XML

- ❖ There are four (4) possible values autowiring can have:

- no
- byName
- byType
- constructor

Inject beans by matching
names defined in config files

Example:

```
<bean id="catalog" autowire="byName" ... />
```

69

Notes:

The byName type of autowiring is presented on the next few slides. It allows Spring to "discover" the appropriate beans to inject by looking at other bean definitions in its configuration. The byType technique causes Spring to look for beans based on their type rather than their names. This technique is rather tricky because Spring can fail to inject a bean properly if two bean definitions exist with the same type. Constructor lets Spring determine how to autowire based on the format of the class' constructor.

Autowiring through XML ...

- ❖ In the following example, a *Catalog*, contains a ***ProductDao*** bean

```
public class Catalog {  
    ProductDao dao;  
  
    public Catalog() {}  
  
    public Catalog(ProductDao dao) { this.dao = dao; }  
  
    public ProductDao getProductDAO() { return dao; }  
  
    public void setProductDAO(ProductDao dao) {  
        this.dao = dao;  
    }  
}
```

Take note of the names used

70

Notes:

In this example, it is not the name of the field, but rather the getter and setter that will be considered by Spring.

Autowiring through XML ...

- ❖ The configuration for autowiring does not require bean references now

```
<beans ...>

    <bean id="catalogBean"
        class="com.company.springclass.beans.Catalog"
        autowire="byName"/>

    <bean id="productDAO" ←
        class="com.company.springclass.beans.ProductDao"
        />

</beans>
```

*The bean is found by looking at other
bean definitions that match the getXXX()
setXXX() pattern within the class.*

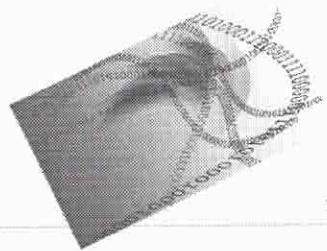
71

Notes:

Spring finds a bean definition with the ID of 'productDAO', which it matches to the setter of the Catalog class and injects it properly. If the setter had been called, setProductDao() instead of setProductDAO() this would have failed.

Working with the Solution

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        ApplicationContext context = new  
            ClassPathXmlApplicationContext("applicationContext.xml");  
        Catalog catalog = (Catalog) context.getBean ("catalogBean");  
  
        catalog.getProductDAO().save (  
            new Product(1, "Socks", 1.89, 3.99, 101));  
    }  
}
```



72

Notes:

The solution presented here is merely for completion of the example and doesn't have any new concepts presented.

Design Consideration: Should you use autowiring?

- ❖ Arguments against autowiring:

- Doesn't scale well
 - Larger projects can run into trouble when unexpected classes are loaded by the container's "guesswork"

- ❖ Arguments for autowiring:

- Reduces XML configuration
 - It *can* scale well with proper cautions and naming conventions

73

Notes:

Autowiring with Annotations

- ❖ In our *Catalog*, **ProductDao** example, the ProductDao is automatically injected by Spring when the @Autowired annotation is used:

```
public class Catalog {  
    ProductDao dao;  
    // ...other methods left out for brevity...  
    @Autowired  
    public void setProductDAO(ProductDao dao) {  
        this.dao = dao;  
    }  
}
```

74

Notes:

Using Annotations

- ❖ To use annotations, your XML config file must indicate this...

```
<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-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd"
>
    <context:annotation-config/>
    <bean id="catalogBean"
          class="com.company.springclass.beans.Catalog"/>
        <bean id="productDAO"
              class="com.company.springclass.beans.ProductDao" />
</beans>
```

Add the context namespace also

75

Notes:

The only major changes needed to use annotation-based configuration is to include the context xml namespace and to specify the `<annotation-config>` xml element.

Design Considerations: Issues with Annotations

- ❖ Arguments against the use of annotations in Spring-based apps:
 - These wirings are configuration metadata that should be kept separate from the source code
 - They depend on proper variable naming conventions, which could change if a team member is not careful

76

Notes:

Summary

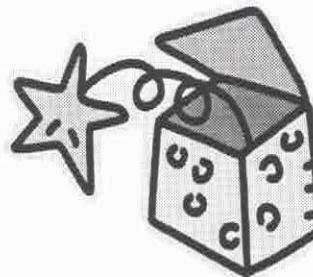
- ❖ Spring's IOC container can inject beans with various scopes into other beans
- ❖ Spring supports several different scopes for beans, however, singleton is still the default
 - It also supports internationalization and placeholders in config files
- ❖ To web-enable a Spring application, be sure to configure the proper listeners

77

Notes:

Lab 2c: Annotations and Autowiring

- ❖ Working from **SpringLab02c**, create a solution that uses both annotations and autowiring.
- ❖ Perform the steps identified in *instructions.html* found in the **Springlab02c project** folder.



78

Notes:

III. Spring JDBC

79

Spring JDBC Overview

- ❖ Classic JDBC vs. Spring JDBC
- ❖ JDBC Template
- ❖ DataSources
- ❖ Executing Statements
- ❖ Processing Results
- ❖ Updates

80

Notes:

What Does Spring JDBC Offer?

- ❖ Spring manages *JDBC Resources*
 - Opening / closing connections
 - Reading JDBC Connection Properties
 - Iterates through ResultSets
 - Manage Transaction Details
 - Handles JDBC Exceptions

81

Notes:

Classic JDBC

- ❖ Given the following Driver



```
public class Driver {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("beans.xml");  
  
        CatalogManager catalog =  
            (CatalogManager) context.getBean("catalog");  
  
        Iterator iter = catalog.getProducts().iterator();  
  
        while(iter.hasNext()) {  
            Product p = (Product)iter.next();  
            System.out.println(p);  
        }  
    }  
}
```

82

Notes:

The following are sample properties that are used to connect to a database.

jdbc.properties

```
jdbc.driverClassName=com.ibm.db2.jcc.DB2Driver  
jdbc.databaseName=jdbc : db2 : SAMPLE  
jdbc.username=db2admin  
jdbc.password=db2admin
```

Classic JDBC ...

```
public class CatalogManagerImplementation implements CatalogManager {  
    public List getProducts() {  
        List products = new ArrayList();  
  
        Connection conn = null;  
        Statement stmt = null;  
        ResultSet rs = null;  
  
        Properties properties = new Properties();  
  
        try {  
            properties.load(new FileInputStream("jdbc.properties"));  
        }  
        catch (IOException e) {  
            System.out.println("An error occurred while reading the properties file.");  
            System.exit(42);  
        }  
  
        try {  
            Class.forName(properties.getProperty("driverName"));  
            conn = DriverManager.getConnection(properties.getProperty("url"),  
                properties.getProperty("username"),  
                properties.getProperty("password"));  
  
            stmt = conn.createStatement();  
            rs = stmt.executeQuery("SELECT * FROM Products");  
        }
```

Problems with Classic JDBC

- Lots of exception handling
- Repeated connection mgmt code
- Lots of work processing results

83

Notes:

```
while (rs.next()) {  
    int productId = rs.getInt("ProductId");  
    String name = rs.getString("Name");  
    String partNumber = rs.getString("PartNumber");  
    String description = rs.getString("Description");  
    double price = rs.getDouble("Price");  
    double cost = rs.getDouble("Cost");  
    int vendorId = rs.getInt("VendorId");  
  
    products.add(new Product(productId, name, description,  
        partNumber, cost, price, vendorId));  
}  
}  
catch (ClassNotFoundException e) {  
    System.out.println("Error loading driver.");  
}  
catch (SQLException se) {  
    System.out.println("Error processing the SQL.");  
}  
finally{  
    if (conn != null) {  
        try {  
            conn.close();  
        }  
        catch (SQLException e) {}  
    }  
}  
}  
return products;  
}
```

83

Some Spring JDBC Classes

- ❖ `JdbcTemplate` (the workhorse of Spring JDBC)
 - `MappingSqlQuery`
 - `SqlUpdate`
- ❖ `JdbcDaoSupport` (a wrapper for Jdbc Template)

84

Notes:

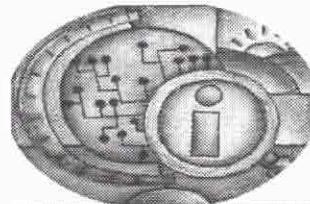
Configuring the DataSource

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <property name="driverClassName"
              value="com.ibm.db2.jcc.DB2Driver"/>
    <property name="url" value="jdbc:db2:SAMPLE"/>
    <property name="username" value="db2admin"/>
    <property name="password" value="db2admin"/>
</bean>
```

The catalog bean is the DAO.
Spring injects the dataSource into it.

```
<bean id="catalog"
      class="com.company.springclass.services.CatalogManagerImplementation">
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
</bean>
```

The Jakarta Commons
DataSource has setters
that Spring can inject
values into it.



85

Notes:

Notice the bean definition for our CatalogManagerImplementation class. It tells Spring to inject the dataSource bean into it. This means our CatalogManagerImplementation class needs to have a setter called setDataSource(). Upon a casual glance at the code in the provided projects, you will notice that it doesn't! So, how does Spring inject the dataSource into this bean? The answer is coming in 2 slides, but a hint for now: the JdbcDaoSupport class.

Jdbc Template

- ❖ *JdbcTemplate* allows you to issue any type of SQL statement and return any type of result
 - It manages a *DataSource* object
 - You configure it from your XML definition file

- ❖ Some ***JdbcTemplate*** methods:

```
List execute(String sql, RowMapper rowMapper)
Map queryForMap(String sql)
Object queryForObject(String sql, RowMapper rowMapper)
public List queryForList(String sql)
public int queryForInt(String sql)
public int update(final String sql)
```

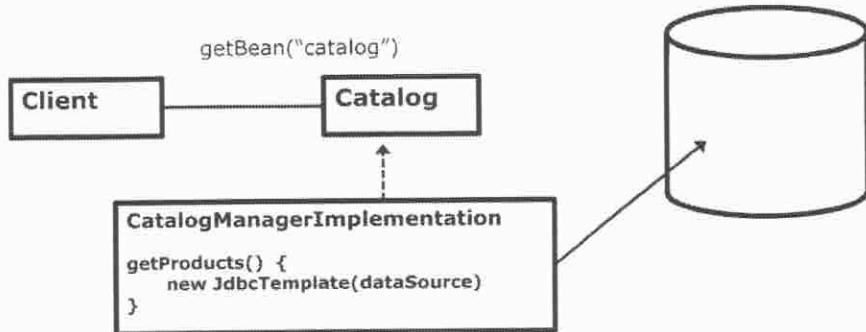
There are over **70** Jdbc Template methods and over 40 to return various query results

86

Notes:

Working with JdbcTemplate

- ❖ *JdbcTemplate* is a concrete class and can be instantiated directly
- ❖ It needs to know which data source will be used



87

Notes:

There are several ways to work with Spring's *JdbcTemplate*. One way is simply to instantiate it, passing the desired data source object to its constructor or calling its *setDataSource()* before using it. In the following example, the client obtains the Catalog service via a familiar `context.getBean()` call. It invokes the *getProducts()* method which, in turn, then obtains all the products from the database. It does this by using a newly instantiated *JdbcTemplate* object. See the next slide.

Getting Data from JdbcTemplate

```
ApplicationContext context = new  
    ClassPathXmlApplicationContext("applicationContext.xml");  
CatalogManager catalog =  
    (CatalogManager) context.getBean("catalog");  
  
Iterator iter = catalog.getProducts().iterator();  
  
while(iter.hasNext()) {  
    Map product = (Map)iter.next();  
    System.out.println(product);  
}  
  
Let's explore this method further ...
```

88

Notes:

This example grabs and displays products from the catalog by retrieving them through the Spring JDBC API. Here you can see the catalog object invoking *getProducts()*. This method retrieves and returns a collection of product objects.

Getting Data from JdbcTemplate ...

```
public class CatalogManagerImplementation  
    implements CatalogManager{  
  
    private DataSource dataSource;  
  
    public void setDataSource(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    public List getProducts() {  
        String sql = "SELECT * FROM springclass.products";  
        return new JdbcTemplate(dataSource).queryForList(sql);  
    }  
}
```

This is the complete solution, but how does the dataSource get set?

89

Notes:

The `getProducts()` method in this example returns a List of Map objects. Each row returned from the ResultSet is placed into a Map object. This may not be the easiest way to work with data, so we'll improve this solution in just a moment.

The `JdbcTemplate` object is passed to the `dataSource` object, but how does it get set? There is a setter method, but it is not used in this code. Where does it get invoked? See next slide...

Getting Data from JdbcTemplate ...

applicationContext.xml

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsq://localhost"/>
    <property name="username" value="student"/>
    <property name="password" value="student"/>
</bean>

<bean id="catalog"
      class="com.company.springclass.services.
          CatalogManagerImplementation">
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
</bean>
```

The datasource is injected into the Catalog Service

90

Notes:

Notice the Spring config file defines the data source. To use it, we can utilize Spring's dependency injection features.

Improving the Solution

- ❖ In the previous example, we had to add a `DataSource` property and instantiate the `JdbcTemplate` ourselves.
- ❖ By having `CatalogManagerImplementation` inherit from `JdbcDaoSupport`, Spring will take care of:
 - Injecting the `DataSource`
 - Instantiating the `JdbcTemplate`



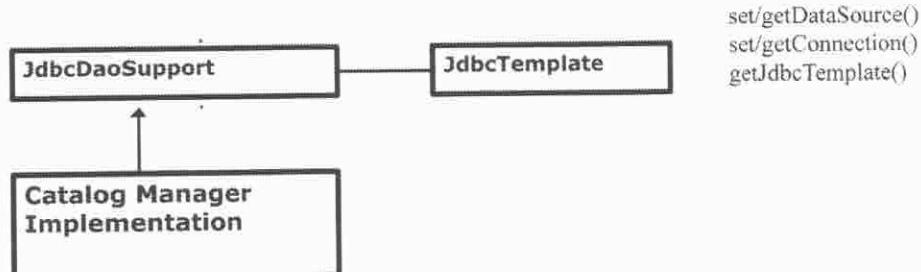
91

Notes:

The `JdbcDaoSupport` class provides a nice way for your application to "hook" into the Spring JDBC classes and then take advantage of its IOC features.

JdbcDaoSupport

- ❖ *JdbcDaoSupport* wraps a *JdbcTemplate* for easy access to it
 - Your DAO should extend **JdbcDaoSupport**
- ❖ The primary **JdbcDaoSupport** methods are:



92

Notes:

Use this class to make Spring JDBC even easier.

Using JdbcDaoSupport

```
public class CatalogManagerImplementation  
    extends JdbcDaoSupport implements CatalogManager {  
    public List getProducts () {  
        String sql = "SELECT * FROM springclass.products";  
        return getJdbcTemplate().queryForList(sql);  
    }  
}
```

No data source or setter
is required now

Instantiating the
JdbcTemplate object is
not necessary now

93

Notes:

In this revised solution, only the CatalogManagerImplementation has changed. Notice it extends JdbcDaoSupport. This provides a convenient `getJdbcTemplate()` method and the data source is automatically injected by Spring and stored in the CatalogManagerImplementation because the parent class has a setter for it.

Improving Repetitive JDBC

- ❖ In JDBC, retrieving a row of data usually involves the same step every time:
 - Take one row from the *ResultSet*
 - Extract the data from it
 - Instantiate a business object and insert the data
- ❖ In Spring, we can simplify this repetitive work by defining this task in a class called a **RowMapper**

94

Notes:

A Row Mapper

```
public class ProductMappingQuery extends MappingSqlQuery
    implements RowMapper {
    public ProductMappingQuery(DataSource ds) {
        super(ds, "SELECT * FROM Products");
        compile();
    }
    public Object mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        int productId      = rs.getInt("ProductId");
        String name        = rs.getString("Name");
        String partNumber  = rs.getString("PartNumber");
        String description = rs.getString("Description");
        double price       = rs.getDouble("Price");
        double cost        = rs.getDouble("Cost");
        int vendorId       = rs.getInt("VendorId");
        return new Product(productId, name, description,
                            partNumber, cost, price, vendorId);
    }
}
```

How to map 1 row

95

Notes:

You can define a class that should inherit from `MappingSqlQuery`. Override its `mapRow()` method and define how an object should be mapped from an SQL row. Return one object from the `mapRow()` method. When implemented, `mapRow()` is called by the Spring Framework for each record in the `ResultSet`, much like a callback. So, if there are ten records retrieved, `mapRow()` is invoked 10 times. Each time the current row in the `ResultSet` is advanced by Spring, a `rowNumber` is also passed to the method for counting purposes, if needed.

Using the RowMapper

- ❖ Spring can now use the *RowMapper* to return collections of business objects rather than collections of Maps
- ❖ Remember that Spring calls *mapRow()*, not you!
- ❖ Spring passes the **ResultSet** into *mapRow()*



96

Notes:

The Spring DAO

```
public class CatalogManagerImplementation extends JdbcDaoSupport  
implements CatalogManager  
{  
    public List getProducts() {  
        return new ProductMappingQuery(getDataSource()).execute();  
    }  
}
```

When the DAO extends JdbcDaoSupport, it inherits a set/getDataSource() method

execute() returns a List of objects (one object is returned from each call to mapRow())

97

Notes:

The ProductMappingQuery object overrides the *mapRow()* method and defines how 1 row from the ResultSet should be processed. Typically, this work is to extract the data and populate a newly instantiated business object.

execute() returns a java.util.List of business objects that can now be processed back in the client.

Obtaining a Single Object

```
catalog.getProduct(1001);
```



This uses the same row mapper as the previous example.

```
public Product getProduct(int productId) {  
    String sql = "SELECT * FROM Products WHERE productId=?";  
    return (Product) getJdbcTemplate().queryForObject(sql,  
        new Object[] {new Integer(productId)},  
        new ProductMappingQuery());  
}
```

queryForObject() takes 3 Parameters

- ✓ The SQL statement
- ✓ Parameters to be inserted
- ✓ A RowMapper

98

Notes:

Here, *queryForObject()* is a method of the JdbcTemplate that retrieves a single object when you specify the SQL, any parameters to pass into the SQL (as an array of objects), and the same *ProductMappingQuery()* object as before.

Spring JDBC Simplifies the Work

- ❖ No closing of Connections, Statements, ResultSets needed
 - Spring manages resources
 - Closing and releasing connections
- ❖ No checked exceptions requiring handling
 - How can you obtain Error messages?



99

Notes:

Spring JDBC Error Handling

- ❖ **Spring JDBC** turns *checked exceptions* into runtime exceptions

- ❑ try-catch blocks are optional, not mandatory
 - ❑ Catch the runtime exceptions, if desired
 - ❑ Configure your own custom exception codes

- ❖ Spring JDBC's root exception is

org.springframework.dao.DataAccessException

- ❑ Since it is a runtime exception, handling it is only required for recovery and logging purposes

100

Notes:

Spring JDBC Error Handling ...

```
public Product getProduct(int productId)
{
    String sql = "SELECT * FROM Products WHERE productId=?";
    Product p = null;

    try {
        p = (Product) getJdbcTemplate().queryForObject(sql,
            new Object[]{new Integer(productId)},
            new ProductMappingQuery());
    }
    catch (DataAccessException e) {
        logger.severe(e.getMessage());
    }

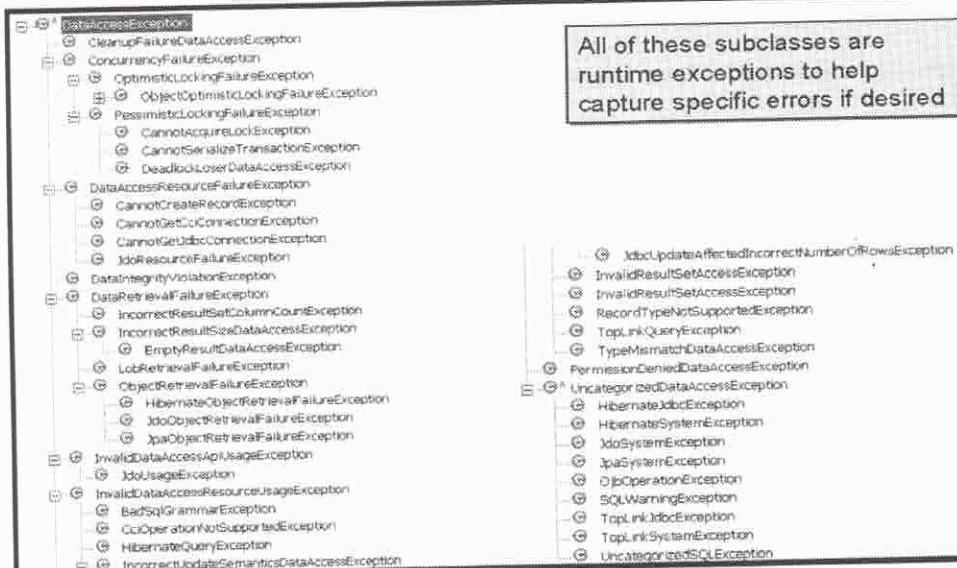
    return p;
}
```

101

Notes:

DataAccessException is a runtime exception. There are a number of subclasses to catch specific error types (see next slide).

DataAccessExceptions



102

Notes:

To trap database specific errors, you can configure them in a Spring XML definition file as follows:

```
<bean id="DB2_9"
      class="org.springframework.jdbc.support.SQLErrorCodes">

    <property name="badSqlGrammarCodes">
        <value>11,24,33</value>
    </property>
    <property name="dataIntegrityViolationCodes">
        <value>1,12,17,22</value>
    </property>
    <property name="customTranslations">
        <list>
            <bean
                class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
                <property name="errorCodes">
                    <value>942</value>
                </property>
                <property name="exceptionClass">
                    <value>com.company.app.exceptions.MyCustomException</value>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

102

Spring JDBC Error Msg Example

```
String sql = "SELECT * FROM Products WHERE prodId=?";  
Product p = null;  
  
try  
{  
    p = (Product) getJdbcTemplate().queryForObject(sql,  
        new Object[] {new Integer(productId)},  
        new ProductMappingQuery());  
}  
catch (DataAccessException e)  
{  
    logger.severe(e.getMessage());  
}
```

This should be productId

```
SEVERE: PreparedStatementCallback; bad SQL grammar [SELECT * FROM  
Products WHERE prodId=?]; nested exception is  
com.ibm.db2.jcc.c.SqlException: DB2 SQL error: SQLCODE: -206,  
SQLSTATE: 42703, SQLERRMC: PRODID
```

103

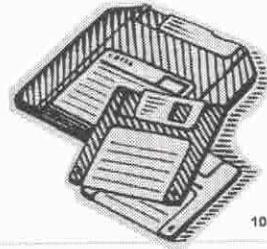
Notes:

Updates

- ❖ Two ways to perform an update/insert/delete
 - Either call the JdbcTemplate's update() method
 - Create a class that extends SQLUpdate, and pre-compile the SQL in its constructor



Both ways are shown...



104

Notes:

These techniques are similar to the ones shown on the previous slides with respect to the retrieval of multiple or single rows of data. In this case, you are inserting rows.

1st Technique: Updates using JdbcTemplate

```
public class OrderManagerImplementation extends JdbcDaoSupport  
implements OrderManager, OrderUpdater {  
  
    public void saveOrder(Order order) {  
  
        String orderSQL = "INSERT INTO ORDERS (ORDERID,  
                                         CUSTOMERID, TOTALPRICE) VALUES (?, ?, ?);  
        getJdbcTemplate().update(orderSQL,  
                               new Object[] {  
                                   new Integer(order.getOrderId()),  
                                   new Integer(order.getCustomerId()),  
                                   new Double(order.getTotalPrice())});  
    }  
}
```

105

Notes:

This technique requires extending the JdbcDaoSupport class so that its *getJdbcTemplate()* method can be invoked.

2nd Technique: Updates via SQLUpdate

```
public class OrderInsert extends SqlUpdate {  
  
    public OrderInsert(DataSource ds) {  
        super(ds, "INSERT INTO Orders (OrderId,  
            CustomerId, TotalPrice) VALUES (?, ?, ?)");  
  
        declareParameter(new SqlParameter(Types.INTEGER));  
        declareParameter(new SqlParameter(Types.INTEGER));  
        declareParameter(new SqlParameter(Types.DOUBLE));  
        compile();  
    }  
    public void insert(Order order) {  
        Object[] params = new Object[] {  
            new Integer(order.getOrderId()),  
            new Integer(order.getCustomerId()),  
            new Double(order.getTotalPrice())};  
  
        update(params);  
    }  
}
```

The OrderManager
invokes this method



106

Notes:

```
public class OrderManagerImplementation extends JdbcDaoSupport  
implements OrderManager, OrderUpdater  
{  
    private DataSource dataSource;  
  
    public void saveOrder(Order order)  
    {  
        new OrderInsert(getDataSource()).insert(order);  
    }  
}
```

Exercise 3: JdbcTemplate

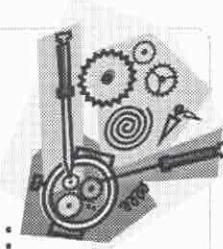
- ❖ Using Spring JDBC techniques, create a solution that can view a single product and all products from the Products table and update a product
- ❖ Use the source files provided in the **SpringLab03 Eclipse** project
- ❖ Work from the additional tasks provided in **instructions.html**



107

Notes:

Summary



- ❖ Spring simplifies JDBC work by using:
 - dependency injection
 - converting checked exceptions into runtime exceptions
 - handling much of the redundant JDBC work
- ❖ Spring offers new classes that simplify the work even further

108

Notes:

IV. DAOs, ORM Frameworks, and Spring

109

J2EE Integration Overview

- ❖ Patterns of Architecture: DAOs
- ❖ Integrating Spring and Hibernate
- ❖ JPA and Spring



110

Notes:

DAO Patterns

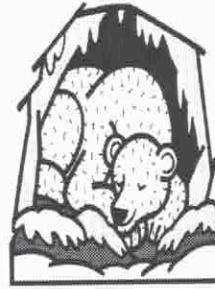
- ❖ *Data Access Objects* wrap repetitive database code hiding the ugly details
 - DAOs can be
 - Generic for an entire application
 - Ex: Class might be called **DAO**
 - Specific for each class
 - Ex: Class might be called **EmployeeDAO**
- ❖ DAOs provide basic CRUD operations...

111

Notes:

Wrapping Hibernate

```
public class EmployeeDAO
{
    private Session session;
    public void create(Employee e) throws DAOException {
        try {
            getSession();
            session.saveOrUpdate(e);
            session.getTransaction().commit();
        }
        catch(HibernateException ex) {
            session.getTransaction().rollback();
            ex.printStackTrace();
        }
        finally {
            if (session != null && session.isOpen()) {
                session.getTransaction().rollback();
                session.close();
            }
        }
    }
}
```



112

Notes:

Wrapping Hibernate ...

```
public void delete(Employee e) throws DAOException
{
    try {
        getSession();
        session.delete(e);
        session.getTransaction().commit();
    }
    catch(HibernateException ex) {
        session.getTransaction().rollback();
        ex.printStackTrace(); // for development only
    }
    finally {
        if(session != null && session.isOpen()) {
            session.getTransaction().rollback();
            session.close();
        }
    }
}
```

113

Notes:

Wrapping Hibernate ...

```
public void update(Employee e) throws DAOException
{
    try {
        getSession();
        session.saveOrUpdate(e);
        session.getTransaction().commit();
    }
    catch(HibernateException ex) {
        session.getTransaction().rollback();
        ex.printStackTrace(); // for development only
    }
    finally {
        if (session != null && session.isOpen()) {
            session.getTransaction().rollback();
            session.close();
        }
    }
}
```

114

Notes:

Wrapping Hibernate ...

```
public Employee find(String id) throws DAOException
{
    Employee e = null
    try {
        getSession();
        e = (Employee) session.load(Employee.class, id);
        session.getTransaction().commit();
    }
    catch(HibernateException ex) {
        session.getTransaction().rollback();
        ex.printStackTrace(); // for development only
    }
    finally {
        if (session != null && session.isOpen()) {
            session.getTransaction().rollback();
            session.close();
        }
    }
    return e;
}
```

115

Notes:

Wrapping Hibernate ...

```
private Transaction getSession() {
    session = HibernateUtils.getSessionFactory().
        getCurrentSession();
    return session.beginTransaction();
}
```

Client code to use the DAO:

```
public static void main (String[] args) {
    Employee e = new Employee("000300", "Smiley");
    e.setFirstName("Jason");
    EmployeeDAO dao = new EmployeeDAO();
    dao.create(e);
    Employee eNew = dao.find(e.getEmpNo());
}
```



116

Notes:

Improving the DAO

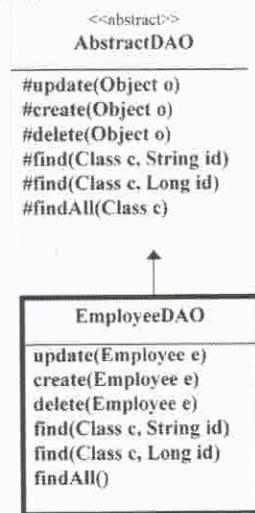
- ❖ Reduce the amount of repetitive code by using the a ***Layered Supertype Pattern***

- ❑ *Patterns of Enterprise Application Architecture*

- Martin Fowler

- ❑ Code is placed in an abstract parent class

- ❑ Concrete child class calls parent methods

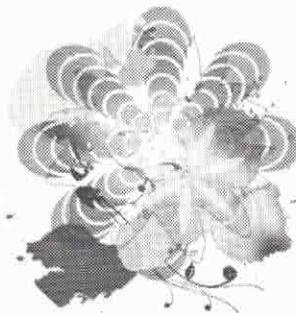


117

Notes:

AbstractDAO

```
public abstract class AbstractDAO {  
    private Session session;  
    protected void create(Object o) throws DAOException {  
        update(o);  
    }  
    protected void update(Object o) throws DAOException {  
        try {  
            getSession();session.saveOrUpdate(o);  
            session.getTransaction().commit();  
        }  
        catch(HibernateException ex) {  
            session.getTransaction().rollback();  
        }  
        finally {  
            close();  
        }  
    }  
}
```



118

Notes:

AbstractDAO ...

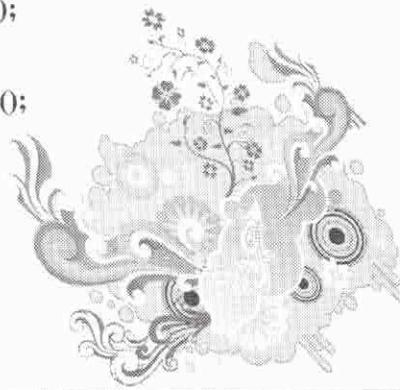
```
protected List findAll(Class classType) throws DAOException {  
    List list = null;  
  
    try {  
        getSession();  
        list = session.createQuery("FROM" +  
            classType.getName()).list();  
        session.getTransaction().commit();  
    }  
    catch(HibernateException ex) {  
        session.getTransaction().rollback();  
  
    }  
    finally {  
        close();  
    }  
    return list;  
}
```

119

Notes:

AbstractDAO ...

```
protected Object find(Class classType, String id)
                      throws DAOException {
    Object o = null;
    try {
        getSession();
        o = session.load(classType, id);
        session.getTransaction().commit();
    }
    catch(HibernateException ex) {
        session.getTransaction().rollback();
    }
    finally {
        close();
    }
    return o;
}
```



120

Notes:

AbstractDAO ...

```
private void close() {  
    if (session != null && session.isOpen()) {  
        session.getTransaction().rollback();  
        session.close();  
    }  
}  
  
private Transaction getSession() {  
    session =  
        HibernateUtils.getSessionFactory().  
        getCurrentSession();  
  
    return session.beginTransaction();  
}
```

121

Notes:

Improved EmployeeDAO

```
public class EmployeeDAO extends AbstractDAO {  
    public void create(Employee e) throws DAOException {  
        super.create(e);  
    }  
    public void delete(Employee e) throws DAOException {  
        super.delete(e);  
    }  
    public void update(Employee e) throws DAOException {  
        super.update(e);  
    }  
    public List findAll() throws DAOException {  
        return super.findAll(Employee.class);  
    }  
    public Employee find(String id) throws DAOException {  
        return (Employee) super.find(Employee.class, id);  
    }  
}
```



122

Notes:

Spring Meets Hibernate

- ❖ Spring JDBC provides a *JDBCTemplate* class
 - It took care of the busy work of handling exceptions, opening and closing connections
- ❖ Spring provides a similar template for Hibernate called **HibernateTemplate**
- ❖ *HibernateTemplate* takes care of:
 - Obtaining the session
 - Handling Exceptions
 - Closing the session
 - Flushing changes
 - Beginning and committing the transaction

123

Notes:

HibernateTemplate is in the **org.springframework.orm.hibernate3** package.

Using Spring's HibernateTemplate

```
Employee e = new Employee("000350","Trayson");
e.setFirstNme("Anna");
```

```
HibernateTemplate ht = new
HibernateTemplate(HibernateUtils.getSessionFactory());

ht.saveOrUpdate(e);

System.out.println(e);
```



Notes:

The HibernateTemplate serves as a wrapper for the Hibernate session.

Configuring Hibernate within Spring's IOC Container

```
beans.xml  
<beans ...>  
  
    <bean id="dataSource"  
          class="org.apache.commons.dbcp.BasicDataSource"  
          destroy-method="close">  
        <property name="driverClassName"  
        value="com.ibm.db2.jcc.DB2Driver"/>  
        <property name="url" value="jdbc:db2:SAMPLE"/>  
        <property name="username" value="db2admin"/>  
        <property name="password" value="db2admin"/>  
    </bean>  
    <!-- continued here next slide -->  
</beans>
```

Hibernate can be integrated with Spring's IOC Container by specifying the Hibernate SessionFactory inside Spring's config file

The datasource can also be mapped here

Hibernate properties may also be configured here



125

Notes:

Configuring the SessionFactory

```
<!-- continued from last slide -->

    <property name="dataSource"><ref local="dataSource"></ref></property>
    <property name="mappingResources">
      <list>
        <value>com/company/springclass/beans/Employee.hbm.xml</value>
        <value>com/company/springclass/beans/Department.hbm.xml</value>
        <value>com/company/springclass/beans/Address.hbm.xml</value>
        <value>com/company/springclass/beans/Customer.hbm.xml</value>
        <value>com/company/springclass/beans/Order.hbm.xml</value>
        <value>com/company/springclass/beans/OrderDetails.hbm.xml</value>
        <value>com/company/springclass/beans/Product.hbm.xml</value>
      </list>
    </property>
    <!-- SessionFactory properties on next slide -->
  </bean>
```

beans.xml



126

Notes:

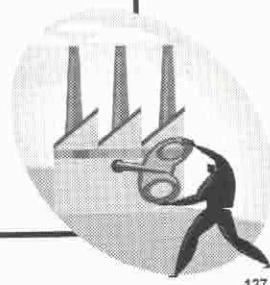
Configuring the SessionFactory ...

```
<!-- SessionFactory properties -->

<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            org.hibernate.dialect.DB2Dialect
        </prop>
        <prop key="current_session_context_class">thread</prop>
        <prop key="cache.provider_class">
            org.hibernate.cache.NoCacheProvider
        </prop>
        <prop key="show_sql">true</prop>
    </props>
</property>

</bean>
```

beans.xml



127

Notes:

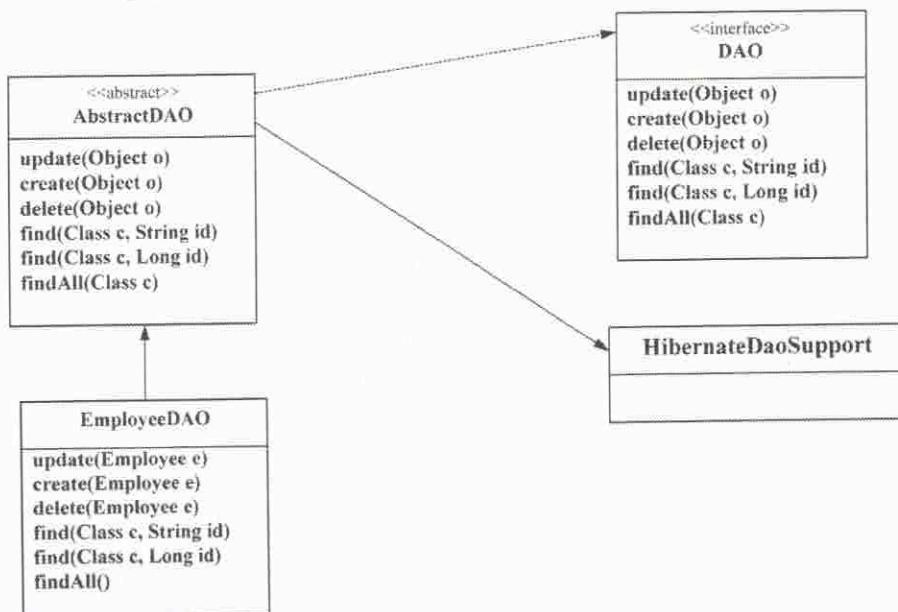
HibernateDaoSupport

- ❖ Spring provides a wrapper for the **HibernateTemplate** called **HibernateDaoSupport**
 - Just like JdbcDaoSupport
- ❖ Modify *AbstractDAO* to extend *HibernateDaoSupport*
 - Use this class to get the **HibernateTemplate** by calling
 - **getHibernateTemplate()**
 - This is very similar to what is done using Spring JDBC

128

Notes:

A New and Improved DAO

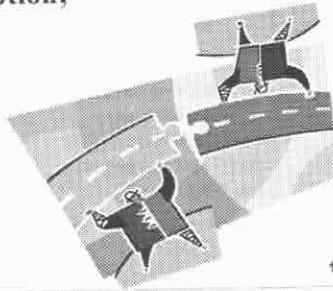


129

Notes:

The DAO Interface

```
public interface DAO
{
    public void create(Object o) throws DAOException;
    public void update(Object o) throws DAOException;
    public void delete(Object o) throws DAOException;
    public Object find(Class c, String id) throws DAOException;
    public Object find(Class c, Long id) throws DAOException;
    public List findAll(Class c) throws DAOException;
}
```



130

Notes:

The New AbstractDAO

```
public abstract class AbstractDAO extends HibernateDaoSupport
    implements DAO {
    public void create(Object o) throws DAOException {
        update(o);
    }
    public void update(Object o) throws DAOException {
        getHibernateTemplate().saveOrUpdate(o);
    }
    public List findAll(Class classType) throws DAOException {
        return getHibernateTemplate().
            find("FROM " + classType.getName());
    }
    public Object find(Class classType, String id)
        throws DAOException {
        return getHibernateTemplate().load(classType, id);
    }
    // other methods shown in footnotes
}
```

131

Notes:

```
public void delete(Object o) throws DAOException {
    getHibernateTemplate().delete(o);
}
```

EmployeeDAO Configuration

- ❖ The *EmployeeDAO* is exactly as before
 - It must be configured in *beans.xml* to use it

```
<bean id="employeeBean"
      class="com.company.springclass.dao.EmployeeDAO">
    <property name="sessionFactory">
      <ref local="sessionFactory"/>
    </property>
</bean>
```

132

Notes:

Client Code

```
Employee e = new Employee("000350", "Trayson");
e.setFirstNme("Anna");

ApplicationContext context =
    new ClassPathXmlApplicationContext("beans.xml");

EmployeeDAO dao =
    (EmployeeDAO) context.getBean("employeeBean");

dao.create(e);
Employee emp = dao.find(e.getEmpNo());
```

Create a transient object

Obtain the DAO through Spring's IOC Container

Work with the DAO

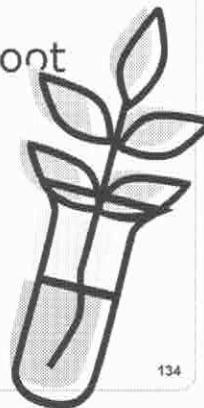


133

Notes:

Lab 4: Hibernate and Spring

- ❖ Working from the **SpringLab04** project, integrate Spring and Hibernate frameworks
- ❖ Follow the specific steps provided in ***instructions.html*** found within the root of the project.



Notes:

JPA Integration

- ❖ Much like Hibernate, JPA can be integrated into a Spring-based application in several different ways:
 - *Use Spring's LocalEntityManagerFactoryBean*

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

- *Obtain the EntityManager via JNDI lookup*

```
<jee:jndi-lookup id=" entityMgrFactory "
                  jndi-name="persistence/myPersistenceUnit"/>
```

135

Notes:

The advantages to the first technique is that it is easy and fast to configure. However, disadvantages with this approach include the fact that an external data source can't be specified this way. The second technique is appropriate for managed environments such as JavaEE app servers that provide JNDI services.

For most efforts, one of these two options will fit most needs.

JPA Integration ...

- ❖ Use Spring's *LocalContainerEntityManagerFactoryBean*

```
<bean id="entityManagerFactory" class=
    "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="loadTimeWeaver">
        <bean class=
            "org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
    </property>
</bean>
```

136

Notes:

This last approach requires the most configuration, but affords the most flexibility, including the ability to provide an externally managed data source, if desired. Implementation using this technique can lead to additional configuration of the classloaders if Tomcat (or WebLogic, or several other app servers) are used.

JpaTemplate

- ❖ As we saw with Spring JDBC and Hibernate, a *JpaTemplate* class provides most of the functionality for interacting with JPA
- ❖ Similar methods to the JPA exist within the *JpaTemplate*:

`find()`
`remove()`
`persist()`
`merge()`

137

Notes:

The `find()`, `remove()`, `persist()`, and `merge()` methods found within the *JpaTemplate* are analogous to the *entitymanager*'s methods with the same name.

JpaDaoSupport

- ❖ Provides a means for easily obtaining the *JpaTemplate*
- ❖ Here's how:
 1. Configure an *EntityManagerFactory* bean within Spring's configuration,
 2. Supply it to your persistence unit name
 3. Pass it into your DAO as a local reference
 4. Invoke **getJpaTemplate()** from within your DAO

138

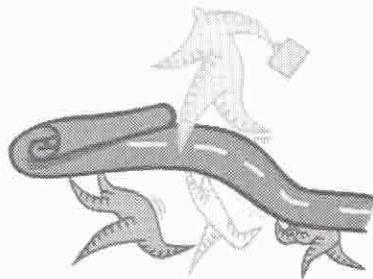
Notes:

The JpaDaoSupport class is the same as the other Support classes. Your DAO should extend this class, making it very easy to manipulate an EntityManagerFactory and retrieve the JpaTemplate object.

Using JpaDaoSupport

❖ Here's how your DAO might look...

```
public class EmployeeDAO extends JpaDaoSupport {  
    public abstract Object find(Object id) {  
        return (Employee) getJpaTemplate().  
            find(Employee.class, new Integer(id.toString()));  
    }  
    public void insert(Object o) {  
        getJpaTemplate().persist(o);  
    }  
    public void update(Object o) {  
        getJpaTemplate().persist(o);  
    }  
    public void remove(Object o) {  
        getJpaTemplate().remove(o);  
    }  
}
```



139

Notes:

Other than the different methods invoked, this DAO is very similar to that of the Hibernate DAO that makes use of the HibernateDaoSupport class from Spring.

Configuring the DAO with JPA

```
<bean id="employeeDAO" class="com.company.crm.dao.EmployeeDAO">
    <property name="entityManagerFactory">
        <ref local="myEntityManagerFactory"/>
    </property>
</bean>

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

140

Notes:

The value, "SpringJPAExample" is the name of the persistence unit found in the META-INF directory of your src folder. This entity manager factory is then passed into the EmployeeDAO via dependency injection. The Employee DAO inherits from JpaDaoSupport and, therefore, inherits a *setEntityManagerFactory()* method. Nothing special needs to happen to your DAO class due to the assistance provided by the JpaDaoSupport class.

Summary

- ❖ Spring and Hibernate can work together
- ❖ SessionFactory, Mapping Resources, DataSources can all be configured through Spring and injected into the Hibernate framework
- ❖ Using Spring's Hibernate support classes, the work associated with mapping objects becomes even easier

141

Notes:

Lab 4b: JPA and Spring

- ❖ Working from the **SpringLab04** project, integrate Spring and the JPA using Hibernate as the underlying provider
- ❖ Follow the specific steps provided in ***instructions.html*** found within the root of the project
- ❖ Create a solution that finds customers by their id by making requests through the *FindService* service and the *CustomerDAO*



Notes:

V. Spring MVC

143

143

Overview

- ❖ MVC Architecture Overview
- ❖ SpringMVC Overview
- ❖ Configuring web.xml
- ❖ Creating a Spring MVC config file
- ❖ Mapping Requests to Controllers
- ❖ Integrating the Service Layer

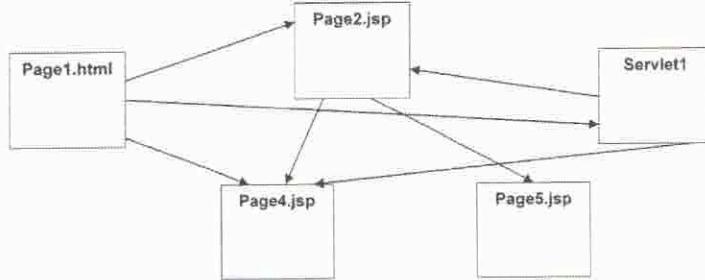


144

Notes:

Traditional Web Architectures

- ❖ In the mid-1990s, many web applications lacked any coherent architecture



- ❖ Difficult to maintain as solutions become more complex

145

Notes:

Early web architectures consisted primarily of static content and were created largely by graphic designers and HTML designers.

Today, many web-based systems are more complex, attempting to perform many more tasks such as authentication, database interaction, and communication with enterprise components. Most web applications now utilize dynamic components such as JSPs, Servlets, and many other resources.

What is MVC?

- ❖ A model-view-controller, or MVC, architecture is a design pattern that makes developing, maintaining, and testing applications easier.
- ❖ The MVC pattern dates back to the late 1970s
 - Originates from the Smalltalk world
- ❖ Today it is heavily implemented
 - Particularly in Java-based web applications

146

Notes:

Model-View-Controller Pattern

- ❖ MVC attempts to separate out
 - Presentation-based tasks from
 - Business domain tasks
- ❖ The View
 - Consists of the application's presentation components:
 - HTML, JSPs, CSSs, client-side scripts, etc.

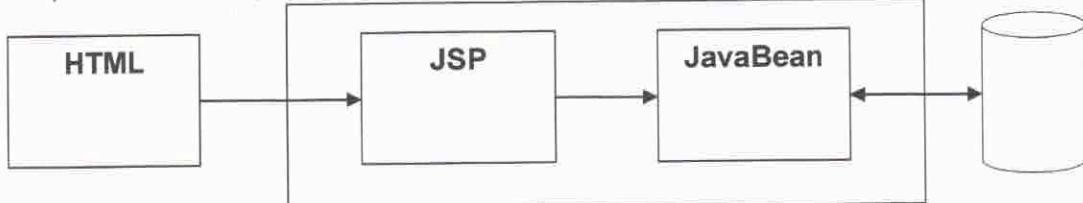
147

Notes:

Larger, more dynamic applications become more difficult to maintain without a well-designed structure in place. The demand for better design techniques led the planners of the JSP specification to present two design options:

- Model 1 Architecture (JSP can handle request processing, i.e. a JSP submits to a JSP)
- Model 2 Architecture (Model-View-Controller approach)

Example Model 1 Design:



Model-View-Controller Pattern ...

❖ The Controller

- Represents the “traffic cop” in the application
- Routes requests to specific tasks (actions)
 - Often consists of a centralized Servlet
 - Ensures proper view document is called
 - Usually has other helpers to assist in work delegation

❖ Model

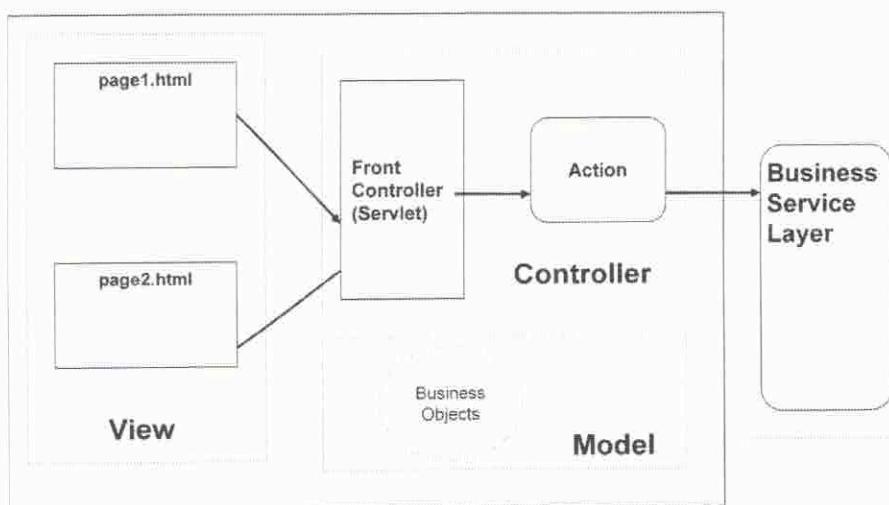
- Business Component Interaction
 - Consists of data components such as JavaBeans (POJOs)

148

Notes:

In a Java-based MVC architecture, the controller is not always a servlet. While this is true in many frameworks like Struts, JSF, and Spring MVC, some frameworks use other components.

Classic MVC Representation



149

Notes:

Why Spring MVC?

- ❖ Spring MVC is a specific MVC implementation like (Struts 1.x/2.x, JSF, Ruby on Rails, Zend)
 - Provides nice integration with Spring beans
 - Loosens the rules components must follow within an MVC framework
 - Easy integration with Spring Web Flows 2

150

Notes:

When looking at MVC architectures, one might be led to say, "Why use Spring MVC?". Certainly Struts, JSF, or countless other web frameworks work nicely with Spring, so why use it? With Spring MVC, the bean configuration is integrated nicely into the solution. Also, controllers have more room for flexibility than do actions in a Struts world.

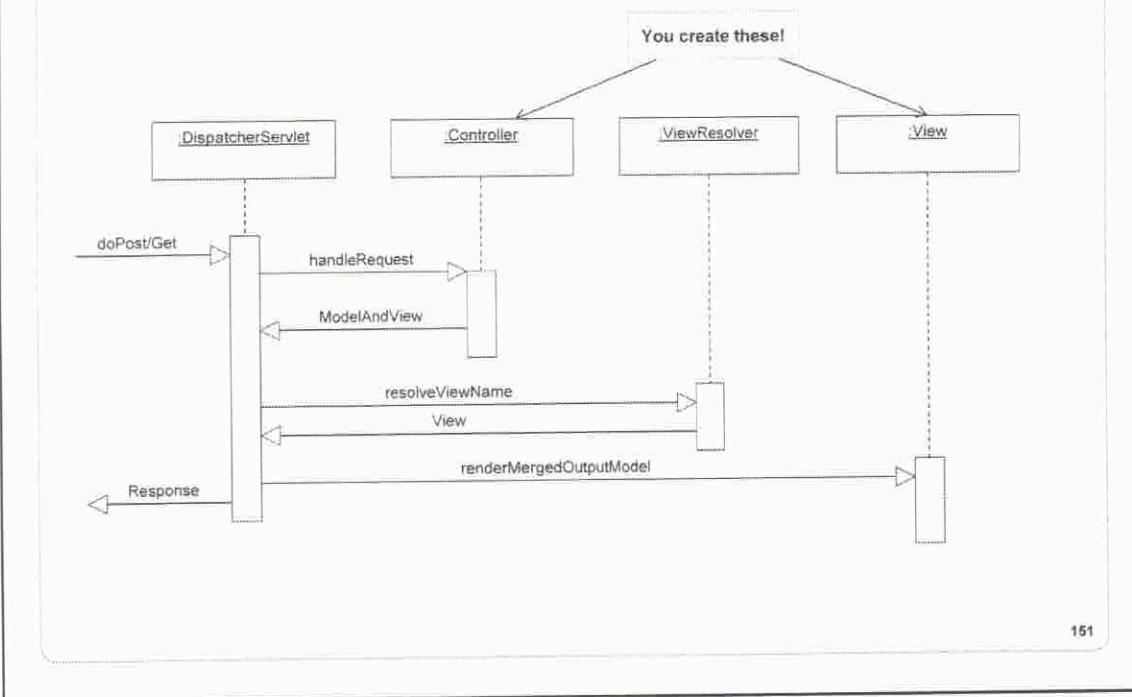
Although there are certainly differences, this course is not going to perform comparisons between the various web frameworks.

Consider this link comparing solutions:

<http://static.raibledesigns.com/repository/presentations/ComparingJavaWebFrameworks-ApacheConUS2007.pdf>

150

Spring MVC Sequence



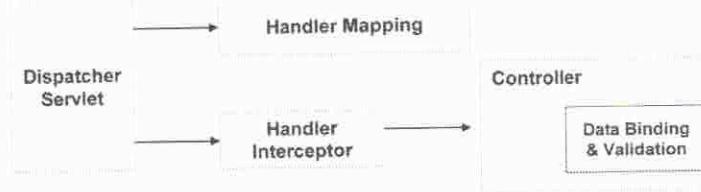
151

Notes:

A Spring MVC application starts and ends a request cycle at the `DispatcherServlet`. It is configured in the `web.xml`. Controllers are defined in another config file and managed by Spring. The `ViewResolver` helps make the decisions on how to traverse from a controller to a view. Finally, the view, usually a JSP, is rendered.

Spring MVC Controllers

- ❖ Controllers are similar to Struts Actions
- ❖ Controllers are implemented as singletons
 - This means they are shared by multiple requests
 - Should be created as thread-safe (stateless)



152

Notes:

The DispatcherServlet utilizes a configured HandlerMapping to determine which Controller to invoke. Before the Controller is invoked a HandlerInterceptor can be invoked (optionally if configured). HandlerInterceptors are similar to Filters in the Servlet API and can perform preprocessing work such as data filtering or authentication.

Creating Controllers

- ❖ Controllers need to implement the *Controller* interface
 - This interface defines a `handlerRequest()` method
- ❖ Extend *AbstractController* instead to get additional basic HTTP features
 - In this case, provide a `handleRequestInternal()` method

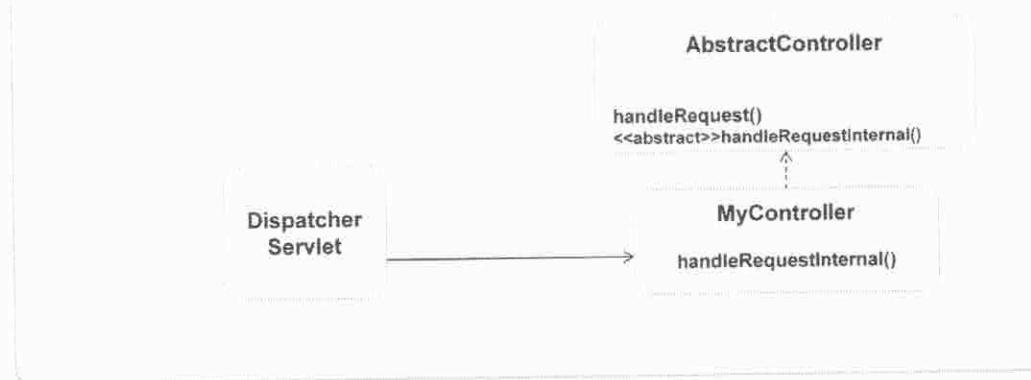
153

Notes:

It is simple to create a Controller by implementing the Controller interface, but if additional features, such as cache-control HTTP headers, access to web objects like request, response, etc. is desired, then it is easiest to extend the *AbstractController* class instead. Depending on the approach taken will determine which method to provide in your controller. The example uses the *AbstractController*.

AbstractController

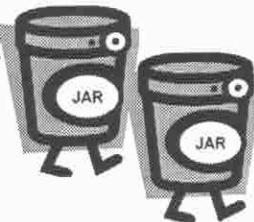
- ❖ AbstractController provides a handleRequest() method invoked by the DispatcherServlet
 - AbstractController in turn invokes handleRequestInternal() which is a method you provide (a form of the Template Method pattern)



Notes:

Establishing the Project

- ❖ Create a web project
- ❖ Add JARs to the lib directory:
 - Add the Spring JARs
 - Jakarta Commons JARs
 - JEE JARs (jstl.jar, standard.jar as needed)



155

Notes:

Configuring web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/emp/*</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

The SpringMVC config file will be named by this value plus **-servlet.xml** appended to it.

All requests with /emp/ in the URL go through SpringMVC DispatcherServlet

156

Notes:

Do not confuse the applicationContext.xml file with the SpringMVC config file. applicationContext.xml defines your business services, spring-servlet.xml defines your SpringMVC mappings.

Important: The name chosen for the servlet-name ('spring' above) becomes the name of your SpringMVC config file with **-servlet.xml** appended onto it. So we should expect to see a **spring-servlet.xml** file in the WEB-INF directory.

156

Mapping URLs to Controllers

- ❖ Spring provided multiple ways to map URLs (requests) to Spring MVC Controllers
 - BeanNameUrlHandlerMapping (default)
 - SimpleUrlHandlerMapping
 - ControllerClassNameHandlerMapping
 - Annotations

157

Notes:

Each of these techniques are useful and easy to work with. The mapping definitions are placed into the SpringMVC config file, which in our case is called **spring-servlet.xml**.

Mapping URLs to Controllers ...

❖ BeanNameUrlHandlerMapping

- ❑ Uses the URL as a bean mapping name directly

The following URL maps to this controller:
<http://localhost:8080/SpringMVC01/emp/me>

```
<bean name="/me" <--  
    class="com.company.springclass.web.controllers.EmployeeController">  
    ...  
</bean>
```

158

Notes:

This technique is the default. Simply map the controller to the URL by providing a bean definition in which the name attribute represents the part of the URL that will uniquely identify which controller to invoke.

Mapping URLs to Controllers ...

❖ SimpleUrlHandlerMapping

- ❑ Uses a map-style approach defining url=controller name pairs

```
<bean  
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
    <property name="mappings">  
        <value>  
            /index.jsp=homeController  
            /page1.jsp=employeeController  
        </value>  
    </property>  
</bean>
```

These values are fixed

These are your url & controller mappings

homeController and employeeController must be names that refer to valid bean definitions elsewhere

159

Notes:

This handler mapping is great when configuring many beans with a one-to-one mapping. A bean id is not needed when using SimpleUrlHandlerMapping. The disadvantage to using it is that it does not really save you much configuration.

It does have a nice feature, the use of * for wildcard mappings. For example /me* will allow /memor or /medep to map to the same controller.

The default is to map within the same servlet mapping, however if the alwaysUseFullPath property is set, then the url must include the full mapping (ex: /emp/index.jsp).

Mapping URLs to Controllers ...

- ❖ ControllerClassNameHandlerMapping
 - This technique uses the concept of "convention over configuration"
 - A url with /me will invoke MeController in the basePackage

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.mvc.support.
      ControllerClassNameHandlerMapping">
    <property name="basePackage">
      <value="com.company.springclass.web.controllers"/>
    </bean>
```

If Spring finds a controller called EmployeeController, this will map automatically to /employee

160

Notes:

There are some additional optional properties that may be specified with a ControllerClassNameHandlerMapping. Here are a few of those properties:

```
<property name="caseSensitive" value="true"/> (url mappings are case sensitive)  
<property name="order" value="0"/>  
<property name="pathPrefix" value="/">
```

Mapping URLs to Controllers ...

- ❖ Annotation-based mapping (Spring 2.5 or later)
 - Uses the @Controller annotation to define controllers
 - Autodetected through classpath scanning
 - Usually used in conjunction with @RequestMapping annotation



161

Notes:

Annotation-based controllers, in conjunction with the use of the RequestMapping annotation is now the preferred technique for handling the mappings between requests and controllers for Spring 3.0.

spring-servlet.xml

```
<beans>
    <!-- view definition -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.
            InternalResourceViewResolver">
        <property name="prefix" value="/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- controller definitions -->
    <bean name="/findEmployee"
        class="com.company.springclass.web.controllers.
            EmployeeController">
    </bean>
</beans>
```

ViewResolvers are explained later

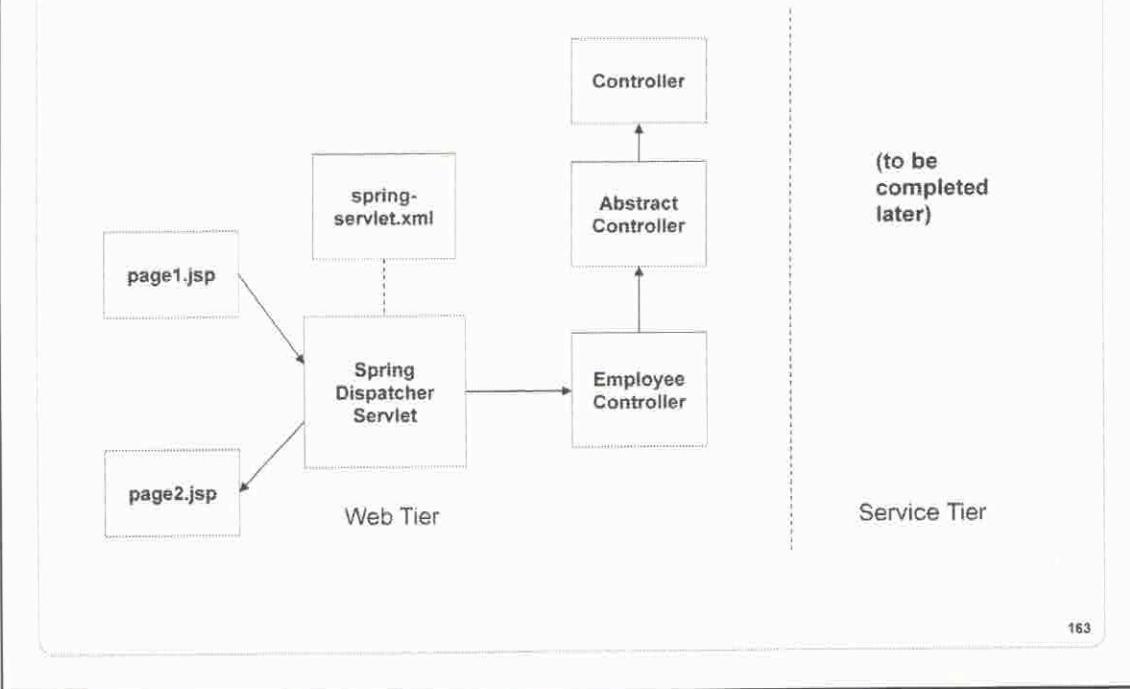
By default, SpringMVC is using a BeanNameUrlHandlerMapping which means it maps a URL to a class. Here, /findEmployee = EmployeeController

162

Notes:

The default HandlerMapping is a BeanNameUrlHandlerMapping. It matches the URL to the name in a bean definition. If a match is found, the specified Controller class is executed.

Our Working Demo

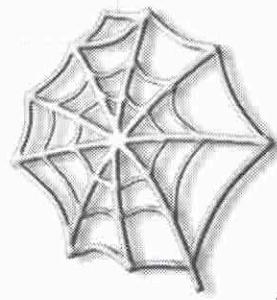


Notes:

Our example starts with a simple request response cycle. A form submits data from `page1.jsp` to the `DispatcherServlet`. The `DispatcherServlet` hands the request to the `EmployeeController`. The `EmployeeController` at this stage simply instantiates an `Employee` object and returns it to `page2.jsp` for display.

web.xml

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/emp/*</url-pattern>
</servlet-mapping>
```



164

Notes:

The two important points here:

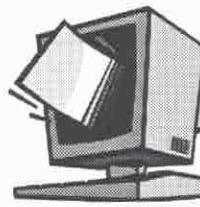
The spring config filename is determined here by using the format: [servlet-name]-spring.xml

The mapping determines which URLs will go through the Spring MVC framework.

page1.jsp

```
<html>
  <head>
    <title>SpringMVC01 Project Employee Finder</title>
  </head>
  <body>
    <h1>Employee Retrieval</h1>
    <h4>Enter an employee to find (100 – 103)</h4>
    <form action="/SpringMVC01/emp/findEmployee">
      Employee ID: <input type="text" name="id" /><br/>
      <input type="submit" value="Find" ></input>
    </form>
  </body>
</html>
```

URL mapping will be /findEmployee



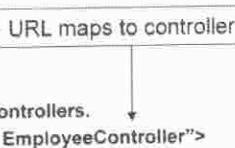
165

Notes:

This page begins the request-response cycle.

spring-servlet.xml

```
<bean>
    <!-- view definition -->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.
            InternalResourceViewResolver">
        <property name="prefix" value="/" />
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- controller definitions -->
    <bean name="/findEmployee"
        class="com.company.springclass.web.controllers.
            EmployeeController">
    </bean>
</beans>
```



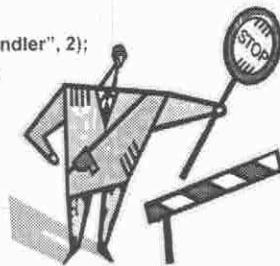
166

Notes:

This config file is using the default handler mapping which maps URLs to Controllers.

EmployeeController

```
public class EmployeeController extends AbstractController {  
  
    protected ModelAndView handleRequestInternal (  
        HttpServletRequest request,  
        HttpServletResponse response) throws Exception {  
        int id = 1;  
        try {  
            id = Integer.parseInt(request.getParameter("id"));  
        }  
        catch (NumberFormatException e) {}  
  
        Employee empl = new Employee (id, "Steve", "Chandler", 2);  
        ModelAndView mv = new ModelAndView("page2");  
        mv.addObject ("employee", empl);  
        return mv;  
    }  
}
```



167

Notes:

This controller identifies its parent as AbstractController. Notice the overridden handleRequestInternal() method. In this method, it is our job to access any services, then define what model components to send to the view and which view to go to. The ModelAndView object performs this for us.

You can use addObject() to add model objects into the page or use the ModelAndView constructor to do this in one step.

page2.jsp

```
<html>
<head>
    <title>SpringMVC01 Project Employee Result</title>

    <style type="text/css">ul { list-style-type: none; }</style>

</head>
<body>
    <h1>Employee Info</h1>
    <ul>
        <li> FirstName: ${ employee.firstName } </li>
        <li> LastName: ${ employee.lastName } </li>
        <li> Dept Id: ${ employee.deptId } </li>
    </ul>
</body>
</html>
```

JSP EL Syntax

168

Notes:

Because the ModelAndView added the empl object (under the key of employee) into the page instance, it can easily be retrieved using JSP expression language syntax.

ModelAndView

- ❖ ModelAndView is a wrapper for any model components to be setn to a view
 - The view can be defined in the constructor
 - A ViewResolver will determine how to map a logical view (like "page2") to and actual view (like page2.jsp)
 - Objects added to the ModelAndView will be injected by spring into the view page instance

169

Notes:

Resolving Views

- ❖ A ViewResolver can be used to determine how to get from Controller to JSP (or whichever appropriate view)
- ❖ A common approach is to use a InternalResourceViewResolver
 - Configure it by defining a prefix and suffix for mappings
 - JSPs can be easily mapped too

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

The web app's
root directory

170

Notes:

A common secure approach is to place JSPs in a directory within the WEB-INF directory. This can be done using a configuration similar to the following:

```
<property name="prefix" value="/WEB-INF/jsp"/>
<property name="suffix" value=".jsp"/>
```

170

VI. MVC Controllers: Using Annotations

171

Spring MVC Annotation Types

- ❖ Can handle multiple actions.
- ❖ Do not need to be stored in a configuration file.
 - Using the **RequestMapping** annotation type, a method can be annotated to make it a request-handling method.
- ❖ Controller Annotation Type

```
@Controller  
public class CustomerController
```

172

Notes:

Spring MVC Annotation Types ...

- ❖ The spring-context schema

```
xmlns:context="http://www.springframework.org/schema/context"
```

- ❖ A <component-scan/> element

```
context:component-scan base-package="basePackage"/>
```

- Integrating in configuration file

```
base-package="com.example.controller"
```

- ❖ A RequestMapping-annotated method

```
@RequestMapping(value = "/customer_input")
```

173

Notes:

Spring MVC Annotation Types ...

- ❖ If more than one attributes appear in @RequestMapping, the value attribute name must be written.
- ❖ The method attribute takes a set of HTTP methods that will be handled by the corresponding method.
`method={RequestMethod.POST, RequestMethod.PUT}`
- ❖ If the method attribute is not present, the request-handling method will handle any HTTP method.

```
@RequestMapping(value="/delete",
    method={RequestMethod.POST, RequestMethod.PUT})
```

174

Notes:

Writing Request-Handling Methods

- ❖ Method arguments

```
myOtherMethod(HttpServletRequest request,  
                Locale locale)
```

- ❖ Argument types

```
org.springframework.ui.Model
```

- ❖ Every time a request-handling method is invoked, Spring MVC creates a Model object and populates its Map with potentially various objects.

175

Notes:

Using An Annotation-Based Controller

- ❖ All requests, including those for static resources, are directed to the dispatcher servlet.

- In order for static resources to be handled properly, some <resources> need to be added

```
<mvc:resources mapping="/css/**" location="/css/">
<mvc:resources mapping="*.html" location="/">
```

- ❖ Without <annotation-driven/>, the <resources/> elements will prevent any controller from being invoked.

176

Notes:

Using An Annotation-Based Controller ...

❖ Controller Class

```
@Controller  
value="/product_input"  
value="/product_save"  
model.addAttribute("product", product);
```

- ❑ The main purpose of having a Model is for adding attributes that will be displayed in the view.
 - In this example, a Product instance was added by calling model.addAttribute

177

Notes:

Using An Annotation-Based Controller ...

❖ The View

```
@import url(css/main.css)

<input type="text" id="name" name="name"
       tabindex="1">

@import url(css/main.css)

${product.name}<br/>
Description: ${product.description}<br/>
Price: $$ {product.price}
```

178

Notes:

Dependency Injection

- ❖ In order for a dependency to be found, its class must be annotated with **@Service**.

```
@Service
```

```
public class ProductServiceImpl implements ProductService
```



179

Notes:

Dependency Injection ...

```
@Controller  
public class ProductController {  
  
    private static final Log logger = LogFactory  
        .getLog(ProductController.class);  
  
    @Autowired  
    private ProductService productService;  
  
    @RequestMapping(value = "/product_input")  
    public String inputProduct() {  
        logger.info("inputProduct called");  
        return "ProductForm";  
    }  
  
    @RequestMapping(value = "/product_view/{id}")  
    public String viewProduct(@PathVariable Long id, Model model) {  
        Product product = productService.get(id);  
        model.addAttribute("product", product);  
        return "ProductView";  
    }  
}
```

180

Notes:

Dependency Injection ...

- ❖ The Spring MVC configuration file has two <component-scan/> elements, one for scanning controller classes and one for scanning service classes.

```
<context:component-scan base-package="moose.controller"/>  
<context:component-scan base-package="moose.service"/>
```

181

Notes:

Redirect and Flash Attributes

- ❖ A forward is faster than a redirect because a redirect requires a round-trip to the server and a forward does not.
 - Redirect to an external site, like a different web site.
 - Cannot use a forward to target an external site so a redirect is the only choice.
- ❖ Avoid the same action from being invoked again when the user reloads the page.
 - If the page is reloaded after the form is submitted, `saveProduct` will be called again and the same product would potentially be added the second time.
 - No side-effect when called repeatedly.
 - For instance, the user could be redirected to a `ViewProduct` page

182

Notes:

Redirect and Flash Attributes ...

- ❖ With a forward, attributes can be added to the Model object and the attributes will be accessible to the view.
 - Spring version 3.1 and later provide a way of preserving values in a redirect by using flash attributes.
- ❖ Must add a new argument of type `org.springframework.web.servlet.mvc.support.RedirectAttributes` in the method.

183

Notes:

Redirect and Flash Attributes ...

```
@RequestMapping(value = "product_save", method = RequestMethod.POST)
public String saveProduct(ProductForm productForm,
    RedirectAttributes redirectAttributes) {
    logger.info("saveProduct called");
    // no need to create and instantiate a ProductForm
    // create Product
    Product product = new Product();
    product.setName(productForm.getName());
    product.setDescription(productForm.getDescription());
    try {
        product.setPrice(Float.parseFloat(productForm.getPrice()));
    } catch (NumberFormatException e) {
    }

    // add product
    Product savedProduct = productService.add(product);

    redirectAttributes.addFlashAttribute("message",
        "The product was successfully added.");
    return "redirect:/product_view/" + savedProduct.getId();
}
```

184

Notes:

Request Parameters and Path Variables

- ❖ Both request parameters and path variables are used to send values to the server.
 - ❑ Both are also part of a URL.
 - ❑ The request parameter takes the form of key=value pairs separated by an ampersand.

`/product_retrieve?productId=3`

185

Notes:

Request Parameters and Path Variables ...

- ❖ An argument that captures request parameter productId.

```
public void sendProduct(@RequestParam int productId)
```

- ❖ A path variable is similar to a request parameter, except that there is no key part, just a value.

```
/product_view/productId
```

- ❖ Using path variables

```
@RequestMapping(value = "/product_view/{id}")
public String viewProduct(@PathVariable Long id, Model model) {
    Product product = productService.get(id);
    model.addAttribute("product", product);
    return "ProductView";
}
```

- ❖ Can use multiple path variables in request mapping

186

Notes:

@ModelAttribute

- ❖ The ModelAttribute annotation type can be used to decorate a Model instance in a method.
- ❖ Instance of it retrieved or created and added to the Model object if the method body does not do it explicitly.
 - Spring MVC will create an instance of **Order** every time this **submitOrder** method is invoked.

```
@RequestMapping(method = RequestMethod.POST)
public String submitOrder(@ModelAttribute("newOrder") Order order,
    Model model) {
    ...
}
```

187

Notes:

@ModelAttribute ...

- ❖ If no key name is defined, then the name will be derived from the name of the type to be added to the Model.
 - Every time the following method is invoked, an instance of **Order** will be retrieved or created and added to the **Model** using attribute key **order**.
- ❖ A method annotated with **@ModelAttribute** will be invoked right before a request-handling method.

```
@ModelAttribute
public Product addProduct(@RequestParam String productId) {
    return productService.get(productId);
}
```

188

Notes:

VII. Bindings and Tags

189

189

Data Binding Overview

- ❖ Data binding is a feature that binds user input to the domain model.
 - HTTP request parameters, which are always of type **String**, can be used to populate object properties of various types.
 - Makes form beans (e.g. instances of **ProductForm**) redundant.
- ❖ Need the Spring form tag library.

190

Notes:

Data Binding Overview ...

```
@RequestMapping(value="product_save")
public String saveProduct(ProductForm productForm,
    Model model) {
    logger.info("saveProduct called");
    // no need to create and instantiate a ProductForm
    // create Product
    Product product = new Product();
    product.setName(productForm.getName());
    product.setDescription(productForm.getDescription());
    try {
        product.setPrice(Float.parseFloat(
            productForm.getPrice()));
    } catch (NumberFormatException e) {
    }
}
```

191

Notes:

Data Binding Overview ...

- ❖ Parse the **price** property in the ProductForm because it was a String and a float was needed to populate the Product's **price** property.
- ❖ Because of data binding, the ProductForm class is no longer needed and no parsing is necessary for the price property of the Product object.
- ❖ Another benefit of data binding is for repopulating an HTML form when input validation fails.

192

Notes:

The Form Tag Library

- ❖ A taglib needs to be added to the page:

```
<%@taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
```

- ❖ The commandName attribute is probably the most important attribute as it specifies the name of the model attribute that contains a backing object whose properties will be used to populate the generated form.
 - If this attribute is present, the corresponding model attribute must be added in the request-handling method that returns the view containing this form.
- ❖ The following form tag is specified in the **BookAddForm.jsp**.

```
<form:form commandName="book" action="book_save" method="post">
...
</form:form>
```

193

Notes:

Controller Code

❖ RequestMapping code in the Controller:

```
@RequestMapping(value = "/book_input")
public String inputBook(Model model) {
    ...
    model.addAttribute("book", new Book());
    return "BookAddForm";
}
@RequestMapping(value = "/book_save")
public String saveBook(@ModelAttribute Book book) {
    Category category = bookService.getCategory(book.getCategory().getId());
    book.setCategory(category);
    bookService.save(book);
    return "redirect:/book_list";
}
```

194

Notes:

Data Binding Example

- ❖ The application will list books, add a new book, and edit a book.
 - The **Book** class and the **Category** class are the domain classes.
 - The tags from the form tag library are used.

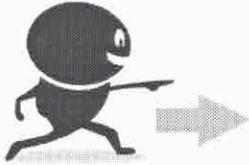


195

Notes:

Data Binding Example ...

The BookList.jsp page:



```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Book List</title>
<style type="text/css">@import url("<c:url value="/css/main.css"/>");</style>
</head>
<body>

<div id="global">
<h1>Book List</h1>
<a href=">Add Book</a>
<table>
<tr>
<th>Category</th>
<th>Title</th>
<th>ISBN</th>
<th>Author</th>
<th>&nbsp;</th>
</tr>
<c:forEach items="${books}" var="book">
<tr>
<td>${book.category.name}</td>
<td>${book.title}</td>
<td>${book.isbn}</td>
<td>${book.author}</td>
<td><a href="book_edit/${book.id}">Edit</a></td>
</tr>
</c:forEach>
</table>
</div>
</body>
</html>
```

196

Notes:

196

Data Binding Example ...

The BookAddForm.jsp page:

```
<%@ taglib prefix="form"
   uri="http://www.springframework.org/tags/form" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Add Book Form</title>
<style type="text/css">@import url("<c:ur.../css/main.css"/>");</style>
</head>
<body>

<div id="global">
<form:form commandName="book" action="book_save" method="post">
<fieldset>
<legend>Add a book</legend>
<p>
<label for="category">Category: </label>
<form:select id="category" path="category.id"
  items="${categories}" itemLabel="name"
  itemValue="id"/>
</p>
<p>
<label for="title">Title: </label>
<form:input id="title" path="title"/>
</p>
<p>
<label for="isbn">ISBN: </label>
<form:input id="isbn" path="isbn"/>
</p>
<p id="buttons">
<input id="reset" type="reset" tabindex="4"/>
<input id="submit" type="submit" tabindex="5"
  value="Add Book">
</p>
</fieldset>
</form:form>
</div>
</body>
</html>
```

197

Notes:

e MODEL ATTRIBUTE

Data Binding Example ...

The BookEditForm.jsp page:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Edit Book Form</title>
<style type="text/css">@import url("<c:uri value="/css/main.css"/>");</style>
</head>
<body>

<div id="global">
<form:form commandName="book" action="/book_update" method="post">
<fieldset>
    <legend>Edit a book</legend>
    <form:hidden path="id"/>
    <p>
        <label for="category">Category: </label>
        <form:select id="category" path="category.id" items="${categories}"
            itemLabel="name" itemValue="id"/>
    </p>
    <p>
        <label for="title">Title: </label>
        <form:input id="title" path="title"/>
    </p>
    <p>
        <label for="author">Author: </label>
        <form:input id="author" path="author"/>
    </p>

```

```

    <p>
        <label for="isbn">ISBN: </label>
        <form:input id="isbn" path="isbn"/>
    </p>
    <p id="buttons">
        <input id="reset" type="reset" tabindex="4">
        <input id="submit" type="submit"
            tabindex="5"
            value="Update Book">
    </p>
</fieldset>
</form:form>
</div>
</body>
</html>
```

198

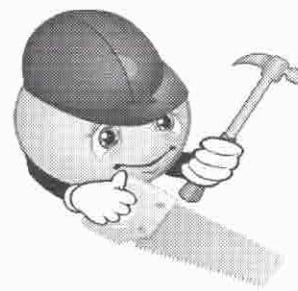
Notes:

VIII. AspectJ and AOP Principles

199

Overview

- ❖ Why AOP?
- ❖ Lots of New Terms!
- ❖ What tools are needed?
- ❖ How Does It Work?



200

Notes:

Why Aspect Oriented Programming?

❖ Problem:

- ❑ Just rolled out a new, flagship, multimillion dollar enterprise application
 - New requirements arise that it must now support



❖ What can you do?

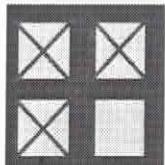
- 1) Jump ship now!
- 2) Get back to work refactoring countless classes to incorporate these capabilities
- 3) Refactor application adding these features in by "weaving" them externally

201

Notes:

Why Aspect Oriented Programming? ...

- ❖ Problem:



*Unneeded
capabilities*

- You are about to use an **architectural model** (e.g. *EJB*) that incorporates features such as persistence, transactions, and security—yet, you do not have the need for all of these features

- ❖ What do you do?

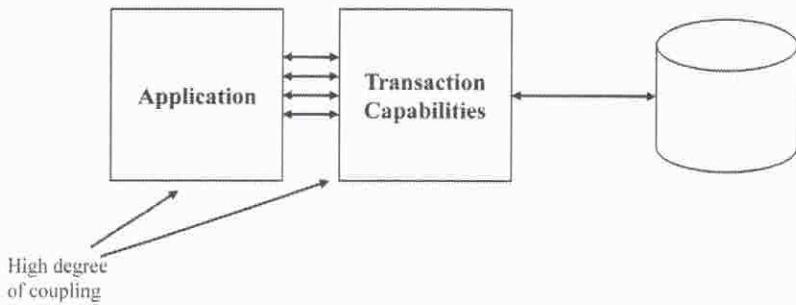
- 1) Continue with an inappropriate model
 - 2) Design an application that modularizes the secondary support features for easy addition or removal of those features

202

Notes:

Why Aspect Oriented Programming? ...

- ❖ AOP is about avoiding the high cost of refactoring applications when requirements change over time



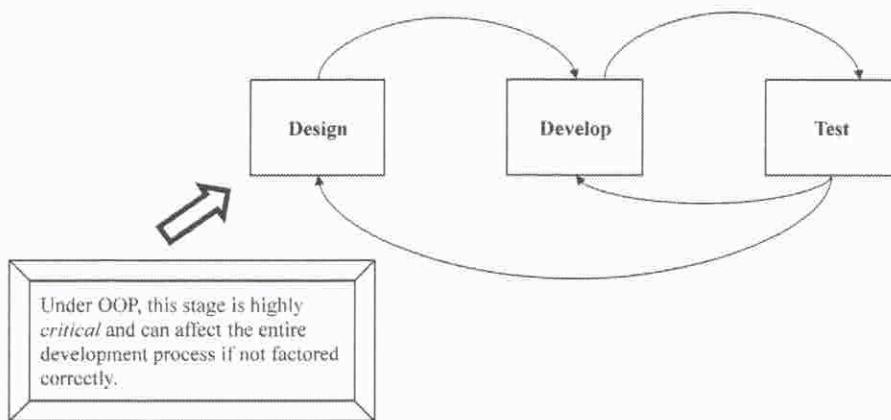
203

Notes:

To incorporate transaction logic into the application, there will be many points throughout the code where transaction-based objects will be invoked. These points cause the dependency on those objects to rise. This makes it difficult to remove those transaction-based objects later on if they are no longer needed.

Why Aspect Oriented Programming? ...

- ❖ Impossible to exactly design an application
 - Possibilities of over design, under design occur



204

Notes:

When a proper model is not developed initially, lots of time can be spent refactoring applications. Classes must be added and removed, methods must be re-written, and code must be rebuilt then retested. This makes the initial design stages very important.

Aspect Oriented Programming

- ❖ Bring us the module!

- ❑ In an AOP application, not only are the core components modularized (as they tend to be in J2EE apps), but *the secondary systems will also be modularized*



First you must understand AOP terminology!...

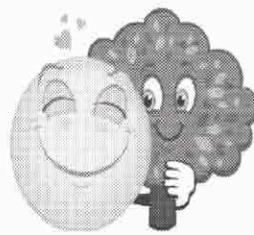
205

Notes:

Core vs. Crosscutting Concerns

❖ **Concerns:**

- ❑ A *concern* is a requirement (subsystem) implemented within a larger application
- ❑ Examples:
 - Transactional systems
 - Logging
 - Persistence
 - Error handling systems
 - Thread Synchronization
 - Caching Operations



206

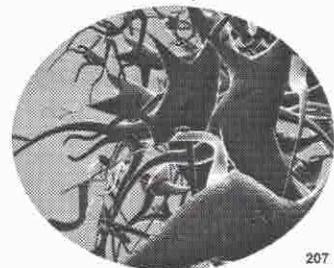
Notes:

There are typically two types of concerns: **core concerns** and **crosscutting concerns**. Core concerns tend to be your primary requirements and your main business services within the application.

Core vs. Crosscutting Concerns ...

❖ **Crosscutting Concerns:**

- ❑ Requirements (subsystems) that interact with several modules within a system
 - *Example: The Human Body*
 - Core concerns are represented by the heart, head, legs, arms. Crosscutting concerns include the nervous system.
 - Our nerves represent a “tangling” of the core components with the nervous system.



207

Notes:

Crosscutting concerns are those concerns that are utilized (referred to) by the core concerns. Core concerns tend to make up the primary, business logic.

Core vs. Crosscutting Concerns ...

❖ *Crosscutting Concerns (cont'd):*

- ❑ *Example: An Enterprise Application* – Core elements include the server-components, business objects, delegates.
- ❑ A logging system would be “tied” into these objects at various points throughout the application.

208

Notes:

Tangling Concerns

- ❖ Code Example illustrating Crosscutting Concerns mixed with Core Concerns (**tangling**)

These are concerns that are not likely a part of the core requirements of this business

```
public class BusinessClass
{
    public void method()
    {
        // perform authorization
        // perform core business logic
        // perform persistence operations
        // log completion of operation
    }
}
```

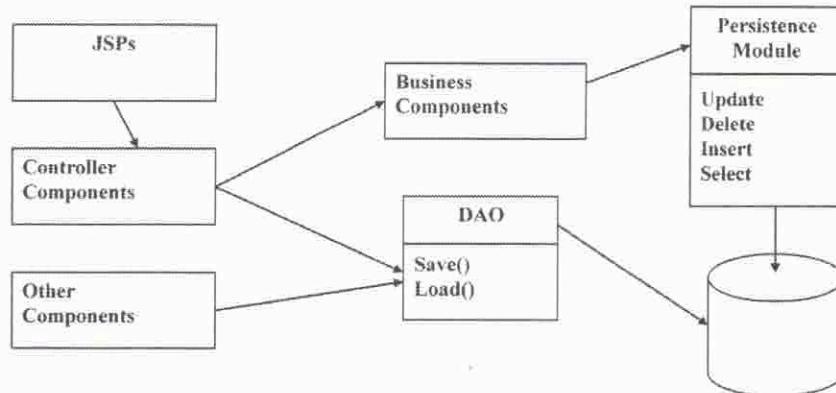
209

Notes:

When various concerns are found implemented within the code of the core components, this is known as **tangling concerns**.

Recognizing Concerns

- ❖ Code Example illustrating Crosscutting Concerns (**scattering**)



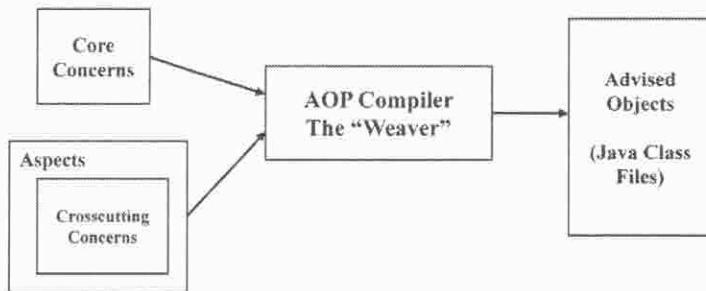
210

Notes:

Another form of implementing crosscutting concerns in conventional systems is via the technique known as **code scattering**. Scattering involves the “spreading” of referenced code throughout the various components within a system. Scattering may lead to repeated code or similarly written code written in various components throughout the system.

Bringing Concerns Together

- ❖ *Core concerns* must eventually implement *crosscutting concerns*
 - In AOP, the concerns are first built separately
 - Then a special compiler, called a “weaver” is used

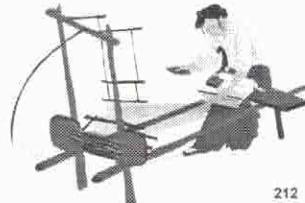


211

Notes:

The Weaver

- ❖ **Weaving** → The process the AOP compiler undertakes by combining core and crosscutting concerns to yield a final component
 - Aspects contain crosscutting concerns
 - They are *weaved* into the core concerns
 - Weaving can be done at compile-time (like AspectJ), or
 - At runtime (like Spring AOP)



212

Notes:

Consider an XSL Transformation as an example. In XSLT, the transformation engine receives a source document that must be converted into a different format on the output. The conversion process is largely dictated by the “rules” that are applied to the transformation. The “rules” in XSLT are found in the XSL files. These “rules” can be easily changed by supplying different XSL transformation files.

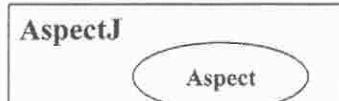
AOP works much the same way, in which the final system will be a result of the original class files (source files) and the “rules” applied during the weaving process. The “rules” in this case are known as **Aspects**.

Aspects Are Like Classes

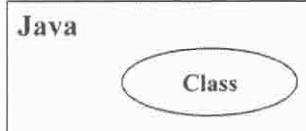
❖ Aspects

- ❑ Modules that identify the rules for “weaving” crosscutting concerns into the system

The structure of an AspectJ aspect is similar to that of a class



analogous to....



213

Notes:

AspectJ is the most well-known implementation of Aspect-Oriented Programming Project for Java. It can be freely obtained from <http://eclipse.org/aspectj>

Aspects

- ❖ *Aspects* are the “rules for weaving”
- ❖ They contain the instructions for “weaving” *crosscutting concerns* into the core system
- ❖ ***Spring Aspects*** can use either
 - SpringAOP (*managed within XML Definition files*)
 - AspectJ (*managed with annotations, e.g. @something*)



Notes:

Spring AOP vs. AspectJ AOP

- ❖ **Spring AOP** represents a limited form of AOP
 - Only a few types of **Join Points** are supported
 - Spring AOP is *invoked at runtime*
 - The runtime advised object is a *proxy object* not a Java class file
- ❖ **AspectJ** has a compiler called **ajc**
 - It *runs at compile-time*
 - Very powerful
 - Supports annotation-based declarations and aspect class structures
 - *Pointcut* and *advice* language is somewhat complex

215

Notes:

Aspect Structure

```
public aspect SomeAspect
{
    // fields
    // methods
    // pointcuts
    // advice
}
```

- ❖ Aspects are composed of:
 - ❑ Declarations
 - ❑ Introductions
 - ❑ Pointcuts
 - ❑ Advice

Aspects may also contain data members and methods much like Java POJOs.

216

Notes:

A Simple Example

```
public class DayOfWeekService
{
    public String getDayOfWeek(int year, int month, int day)
    {
        Calendar c = new GregorianCalendar(year, month, day);
        return new SimpleDateFormat("EEEE").format(c.getTime());
    }
}
```

```
public class Driver
{
    public static void main(String[] args)
    {
        String dayOfWeek =
            new DayOfWeekService().getDayOfWeek(2009, 9, 16);
        System.out.println(dayOfWeek);
    }
}
```

217

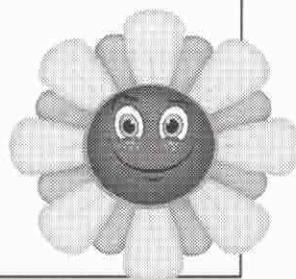
Notes:

A Simple Example

```
public aspect DayOfWeekAspect
{
    pointcut call_GetDayOfWeekPointcut() :
        call(* DayOfWeekService.getDayOfWeek(..));

    before() : call_GetDayOfWeekPointcut()
    {
        System.out.println("In the before-advice...");
    }

    after() : call_GetDayOfWeekPointcut()
    {
        System.out.println("In the after-advice...");
    }
}
```

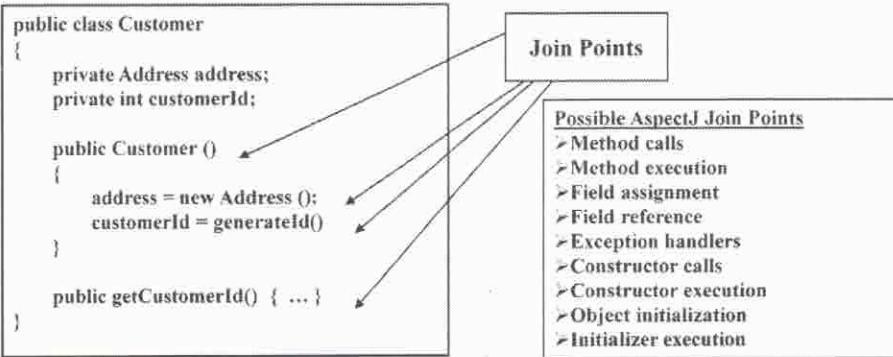


218

Notes:

Join Points

- ❖ *Join Points* are locations within source code where aspects may specify behavior
 - Points within core concerns where the **weaver** can modify your code



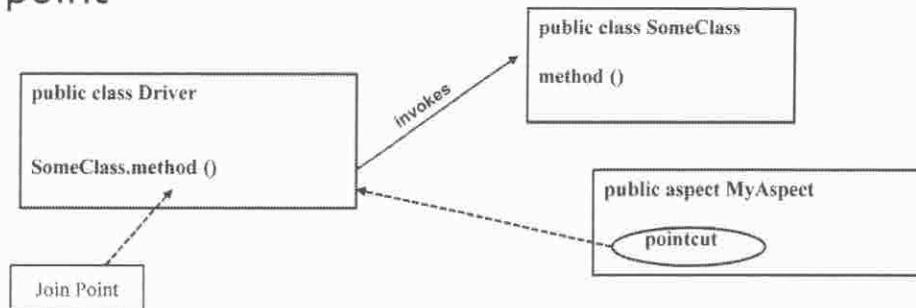
219

Notes:

In Spring AOP, only the execution of methods can be join points.

Pointcuts

- ❖ Structures within an Aspect that select a join point



Pointcut Syntax:

```
pointcut pointCutName(args) : pointcut_designators;
```

Example:

```
pointcut captureMethod() : call(* SomeClass.method(..));
```

220

Notes:

Pointcuts are found within aspects and capture join points. A pointcut can be thought of as the “rule” to be applied and the join point is the condition where that rule would be invoked.

Pointcut Designators

- ❖ Defines what "*activities*" the AOP compiler should look for in the source
- ❖ Types of designators (activities) include

```
call( method signature )
execution( method signature )
get( method signature )
within( method signature )
target( type name )
this( type name )
cflow( join point designator )
handler( exception name )
```

- ❖ Example

```
pointcut call_GetDayOfWeekCall() :
    call(* DayOfWeekService.getDayOfWeek(..));
```

221

Notes:

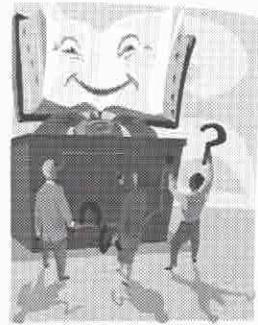
This example identifies a pointcut called 'call_GetDayOfWeekCall'. It tells the AOP compiler to identify any call to methods called *getDayOfWeek()* within the *DayOfWeekService* class that accept any number of parameters and return any types.

Advice

- ❖ **Advice** is the specific instructions to be applied when the AOP compiler selects a pointcut
 - It can be applied *before*, *after*, or *around* (during) the execution of a join point

Example:

```
after() : captureMethod()  
{  
    System.out.println("Class2.method() completed!");  
}
```

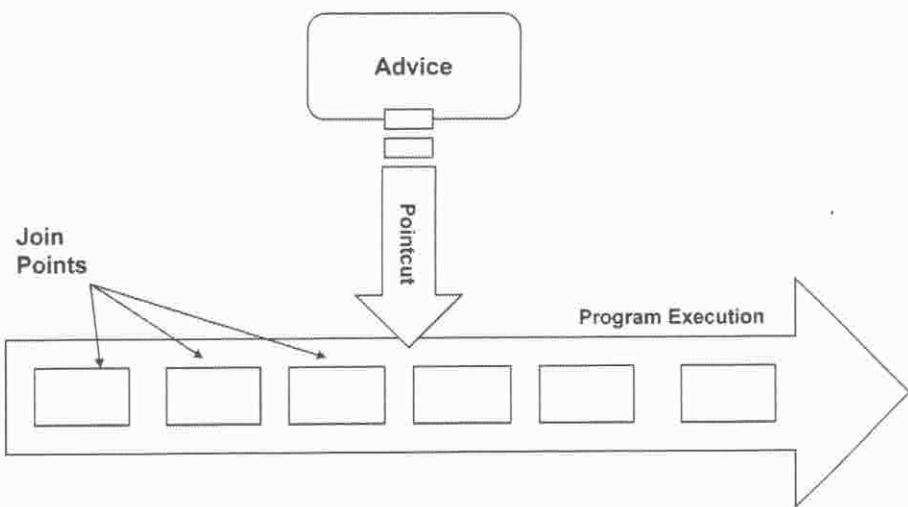


222

Notes:

Advice can be called **before** the execution of a join point's code, **after** the execution of a join point, or **around** it. Around advice can skip, replace, or modify the parameters of the code representing the join point.

Applying Advice



223

Notes:

Introductions

- ❖ **Introductions** are instructions that can make static additions to a class

```
public aspect CustomerAspect
{
    private String Customer.prefix = "Mr.";

    public void Customer.prefixName ()
    {
        return prefix + " " + getName ();
    }

    ...
}
```

Introductions – addition of an instance field and method to the Customer class

The diagram shows two arrows originating from the annotations 'Customer.' and 'Customer.prefix' in the code, pointing to their respective locations in the code block.

224

Notes:

Notice that the aspect (CustomerAspect) is adding a new instance variable and method to the customer class. This is known as static crosscutting.

Additional forms of static crosscutting include:

- ✓ modifying the type-hierarchy
- ✓ softening exceptions
- ✓ declaring compile-time error/warnings

Summary

- ❖ AOP provides a means for easily inserting new requirements into an application
 - *Crosscutting concerns* are great examples where AOP can improve development lifecycles
 - AOP has **two** learning curves:
 - Numerous new terms, such as *Pointcuts*, *Advice*, and *Introductions*
 - A tricky syntax for specifying *pointcut designators*

225

Notes:

Lab 9: Exploring Aspects

- ❖ In this exercise you will create and implement an *Aspect* within an *AspectJ project*
 - ❑ Follow the steps below to install **AspectJ** into Eclipse.
 - ❑ Create the *DayOfWeekAspect* aspect (*File* → *New* → *Other* → *AspectJ* → *Aspect*)
 - ❑ Use *before* and *after* advice on the *DayOfWeekService* class methods
 - ❑ Follow the additional instructions in **SpringLab9a**



Notes:

Installing AspectJ. AspectJ can be installed directly from within Eclipse by selecting:

Help → Software Updates... → Available Software (tab) → Add Site... (button)

Then type in the following URL:

<http://download.eclipse.org/tools/ajdt/34/update>

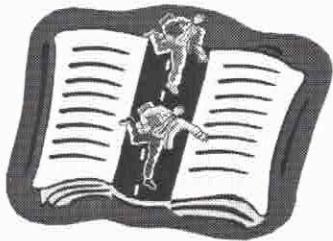
Next, in the list of available software items, select the AJDT Update Site item and then choose Install...

IX. Spring AOP

227

Overview

- ❖ Spring AOP Advice Types
- ❖ Implementing *Before* Advice
- ❖ *After* Advice
- ❖ Using *Around* Advice
- ❖ Creating and Implementing Pointcuts
- ❖ Introductions



228

Notes:

Spring AOP

- ❖ Spring only supports *method joinpoints*
 - AspectJ supports both method and field joinpoints
 - Field joinpoints allow advice to be fired on field modifications
 - This is not a focus of Spring's service architecture
- ❖ Main types of Spring AOP advice
 - *Before* -executes before a method is called
 - *Around* -executes when a method is called
 - *After* -executes after a method is called
 - *Throws* -executes after method throws an exception
 - *Introduction* -modify classes by adding methods and fields

229

Notes:

Spring AOP ...

- ❖ Implement predefined interfaces for each type

Before org.springframework.aop.MethodBeforeAdvice

Around org.aopalliance.intercept.MethodInterceptor

After org.springframework.aop.AfterReturningAdvice

Throws org.springframework.aop.ThrowsAdvice

Introduction org.springframework.aop.IntroductionInterceptor

230

Notes:

There are 2 steps in implementing AOP features within Spring:

1. Create a class that implements the appropriate Advice interface and implements its methods.
2. Configure the classes in the context file.

230

Before Advice Example

- ❖ Simple Example showing Spring *Before Advice*
 - ❑ Implement logging cross-cutting concerns by logging the name of each employee used in **EmployeeDAO** methods (update, create, and delete methods)
 - ❑ This example is interesting because it never requires us to modify the original EmployeeDAO class!

231

Notes:

Implement MethodBeforeAdvice

```
public class LoggingBeforeAdvice implements MethodBeforeAdvice
{
    public void before (Method method, Object[] params,
                       Object target) throws Throwable
    {
        String methodName = method.getName();
        if (params[0] != null)
        {
            Employee e = (Employee) params[0];
            System.out.println("BEFORE ADVICE - Method: " + methodName
                               + ", Employee: " + e.getFirstName() + " " + e.getLastName());
        }
    }
}
```

target.method(params) of
the method being advised.

This example logs (outputs) a message whenever a method
of EmployeeDAO is called.

232

Notes:

Configure the Advice

```
<bean name="loggingBeforeAdvice"
      class="com.company.springclass.aop.LoggingBeforeAdvice" />

<bean id="beforeLogger"
      class="org.springframework.aop.framework.autoproxy.
          BeanNameAutoProxyCreator">
    <property name="interceptorNames">
      <list>
        <value>loggingBeforeAdvice</value>
      </list>
    </property>
    <property name="beanNames">
      <value>employeeDAO</value>
    </property>
  </bean>
```

Our Advice Class

Spring bean to help create the proxy object

Identifies the Advice class(es)

Beans to apply the advice to

233

Notes:

Running the Client

- ❖ The client code remains unchanged (nearly)

```
Employee e = new Employee("000350", "Trayson");
e.setFirstNme("Anna");

ApplicationContext context =
    new ClassPathXmlApplicationContext("beans.xml");

DAO dao = (DAO) context.getBean("employeeDAO");
dao.create(e);
```

The bean returned is a proxy, not EmployeeDAO so you must specify the interface.

```
Employee emp = (Employee) dao.find(Employee.class, e.getEmpNo());

System.out.println(emp);
```

234

Notes:

Because we are now using a proxy, we must use the interface name. The proxy object generated implements this interface, but doesn't extend or modify the specific concrete class (EmployeeDAO).

The output will be as follows:

BEFORE ADVICE -Method: create, Employee: Anna Trayson

Employee: 000350 Anna Trayson

After Advice

- ❖ Creating a solution to act *after* a method returns is useful for the EmployeeDAO **find()** method
 - This example logs the employee name *after* the **find()** method returns



235

Notes:

Implement AfterReturningAdvice

```
public class LoggingAfterAdvice implements AfterReturningAdvice {  
  
    public void afterReturning(Object retValue, Method method,  
        Object [] args, Object target) throws Throwable {  
  
        String methodName = method.getName();  
  
        if (methodName.equals("findAll")) ← Our solution filters out findAll()  
            return; methods.  
  
        if (retValue != null) {  
            Employee e = (Employee) retValue;  
            System.out.println("AFTER ADVICE - Method: " +  
                methodName + " Employee: " + e.getFirstName () +  
                " " + e.getLastName());  
        }  
    }  
}
```

This example logs (outputs) the return value of the find() method

236

Notes:

It is possible to specify exactly which methods should be advised within a class. However, this involves more work, creating or utilizing Pointcut classes to make those decisions.

Configuring the Advice

```
<bean name="loggingAfterAdvice"
      class="com.company.springclass.aop.LoggingAfterAdvice" />

<bean id="afterLogger"
      class="org.springframework.aop.framework.autoproxy.
          BeanNameAutoProxyCreator">
    <property name="interceptorNames">
      <list>
        <value>loggingAfterAdvice</value>
      </list>
    </property>
    <property name="beanNames">
      <value>employeeDAO</value>
    </property>
  </bean>
```

Our Advice Class

Spring bean to help create the proxy object

Identifies the Advice class(es)

Beans to apply the advice to

237

Notes:

Around Advice

- ❖ *Around* advice controls whether the target method is actually invoked
 - Implement the **MethodInterceptor** interface
 - Implement an **invoke()** method in your advice class
 - Call **MethodInvocation** interface's **proceed()** method to invoke the target method if desired
 - It could be used as a *validation technique* by cross-cutting validation logic
 - You would never have to touch application code to implement this feature

238

Notes:

Implement MethodInterceptor

```
public class EmployeeValidationAdvice implements MethodInterceptor {  
    public Object invoke(MethodInvocation invoker) throws Throwable {  
        Object o = null;  
  
        System.out.println("AROUND ADVICE – validating Employee");  
  
        if (invoker.getMethod().getName() . equals ("find")) {  
            String empNo = (String) invoker.getArguments() [1];  
            if (empNo != null && empNo.length() == 6) {  
                try {  
                    Integer.parseInt(empNo);  
                    o = invoker.proceed();  
                } catch (NumberFormatException e) {}  
            }  
        }  
        return o;  
    }  
}
```

If your validation criteria
are satisfied invoke the
target method

Invoke() is called by the Spring AOP framework
after configuring it either programmatically or
declaratively

239

Notes:

Here a class is used to validate the data passed into the *find()* method of the EmployeeDAO object. In order for this to work, Spring must be told which beans to intercept method calls on and also which MethodInterceptor class' *invoke()* method to call in those circumstances.

In the Spring config file, the following statements would be required:

```
<bean id="aroundValidation"  
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">  
    <property name="interceptorNames">  
      <list>  
        <value>employeeValidationAdvice</value>  
      </list>  
    </property>  
    <property name="beanNames">  
      <value>employeeDAO</value>  
    </property>  
</bean>
```

and the bean definition:

```
<bean name="employeeValidationAdvice"  
      class="com.company.springclass.aop.EmployeeValidationAdvice" />
```

Pointcuts

- ❖ *Pointcuts* allow you to configure which methods will be advised
 - Two ways Spring can match methods to see if they should be called:
 - *Static* -advice applied only one time upon first use
 - *Dynamic* -Spring checks each method call within an advised object to see if it should be called
- ❖ When performance is a consideration, you should only use static pointcuts



240

Notes:

To set a pointcut to be static, when creating your Pointcut implementation, override the *isRuntime()* method and return false from it.

Spring-Provided Pointcuts

- ❖ To inform Spring exactly which methods within a class to advise, use a *Pointcut*
- ❖ Do this in 1 of 2 ways:

- ❑ Use a Spring-provided pointcut
 - ❑ Create your own pointcut implementation

`org.springframework.aop.support.NameMatchMethodPointcut`

`org.springframework.aop.support.StaticMethodMatcherPointcut`

`org.springframework.aop.support.DynamicMethodMatcherPointcut`

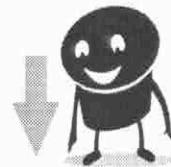
`org.springframework.aop.support.ControlFlowPointcut`

`org.springframework.aop.support.JdkRegexpMethodPointcut`

241

Notes:

Creating Pointcuts



- ❖ To create and use a pointcut:
 - Create a class to implement the specific pointcut interface
 - Override its appropriate methods
 - Either programmatically or declaratively configure it
- ❖ The following example will use pointcuts to call only the **find()** method of EmployeeDAO

242

Notes:

Creating the Pointcut

```
public class IncludeFindOnlyPointcut extends StaticMethodMatcherPointcut
{
    public boolean matches(Method method, Class targetClass)
    {
        return ("find".equals(method.getName()));
    }

    public ClassFilter getClassFilter()
    {
        return new ClassFilter()
        {
            public boolean matches(Class c) {
                return (c == EmployeeDAO.class);
            }
        };
    }
}
```

Define which methods to advise

Define which classes to advise

243

Notes:

This example would invoke the around advice defined in the Spring config file only when the find() method of the EmployeeDAO class is invoked.

Configure the Advisor

```
<bean id="aroundValidation"
    class="org.springframework.aop.framework.autoproxy.
        DefaultAdvisorAutoProxyCreator">
</bean>

<bean id="advisor" class="org.springframework.aop.support.
    DefaultPointcutAdvisor">
    <property name="pointcut">
        <bean class="com.company.springclass.aop.
            IncludeFindOnlyPointcut"/>
    </property>
    <property name="advice">
        <bean class="com.company.springclass.aop.
            EmployeeValidationAdvice"/>
    </property>
</bean>
```

244

Notes:

To set up the pointcut, modify the *aroundValidation* entry in the config file to use a *DefaultAdvisorAutoProxyCreator* instead of the *BeanNameAutoProxyCreator*. Also, set up the Spring *DefaultPointcutAdvisor* with two bean properties: 1) the name of your class that serves as the pointcut; 2) the name of your class that serves as the advice.

Using the Pointcut

- The client code remains untouched, the Advice class no longer checks to see if the find method is being called:

```
Employee e = new Employee ("000350", "Trayson");
e.setFirstName("Anna");

ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

ProxyFactory pf = new
ProxyFactory(context.getBean("employeeDAO"));
DAO dao = (DAO) pf.getProxy();

Employee emp = (Employee) dao.find(Employee.class, e.getEmpNo());

dao.findAll(Employee.class);
```

This method will be advised

This one will not

245

Notes:

The EmployeeValidationAdvice class no longer contains a check to see if the *find()* method is being called:

```
public class EmployeeValidationAdvice implements MethodInterceptor
{
    public Object invoke(MethodInvocation invoker) throws Throwable
    {
        Object o = null;

        System.out.println("AROUND ADVICE -validating Employee number");
        System.out.println("Method called: " + invoker.getMethod().getName());

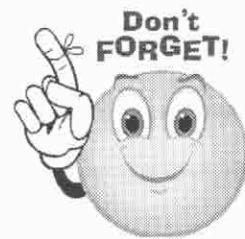
        String empNo = (String) invoker getArguments()[1];
        if (empNo != null && empNo.length() == 6)
        {
            try {
                Integer.parseInt(empNo);
                o = invoker.proceed();
            }
            catch(NumberFormatException e) {}

        }

        return o;
    }
}
```

Summary

- ❖ **Spring AOP** can be used to *apply advice* to classes before, around, and after methods execute
- ❖ Using pointcuts, you can specifically tell Spring how to apply advice at the method level



246

Notes:

Lab 9b: Spring AOP

- ❖ Complete the example that *performs validation using around advice* with Spring AOP
- ❖ Work from the Eclipse project called **SpringLab09b** in the student files directory
- ❖ Add declarations in the Spring config file (*beans.xml*) to implement around advice.
- ❖ Complete the class **EmployeeValidationAdvice** in the **com.company.springclass.aop** package
- ❖ Follow **instructions.html** for more specific tasks

247

Notes:

This page intentionally left blank!



248

248

X. JUnit

249

JUnit 4 Features

- ❖ JUnit supports unit testing by simplifying the creation of TestCases and TestSuites
- ❖ Test methods prior to JUnit4 required prefixing word "test"

```
public void testUpdateCustomer() { ... }
```

- No longer a requirement but still commonly done

250

Notes:

JUnit 4 Features ...

- ❖ JUnit 4 core package is ***org.junit***
 - *junit.framework* older package still provided for backward compatibility
 - Test classes no longer need to extend *junit.framework.TestCase*
 - JUnit4 now uses **annotations** for defining support

251

Notes:

JUnit 4 Annotations

- ❖ `@Test` Annotations are used to define test methods
 - Methods no longer need to begin with testXXX
 - At least one `@Test` annotation is required
- ❖ `@Before` on a method is used as the `setup()` method to perform any initialization
- ❖ `@After` on a method is used as the `tearDown()` method

252

Notes:

JUnit 4 Test Methods

- ❖ Test methods should still return void and accept no parameters
- ❖ To ignore or skip a test, supply a *@Ignore* annotation
- ❖ No Swing or AWT GUI is used with JUnit 4, only a text-based test runner



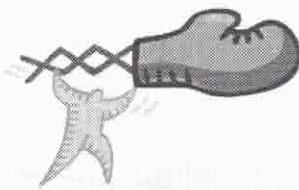
253

Notes:

JUnit 4 Assertions

- ❖ Use *Assert.assertEquals()* since the methods are not inherited from TestCase
 - Alternately, use Java 5 static import technique and use the *assert()* statements as you always have:

```
import static  
org.junit.Assert.*;
```



254

Notes:

Setting Up JUnit 4.5 or Later

❖ Make sure the following two JARs are on the classpath:

- ❑ *junit-4.5.jar* (or a later version)
- ❑ *org.springframework.test.jar*



255

Notes:

Spring-test.jar contains the Spring-based annotations for unit testing. A JUnit 4 or later version jar is required for annotation-based unit testing.

Creating a Spring-based TestCase

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
    locations={"classpath:/applicationContext.xml"})
public class EmployeeServiceTest {
    private EmployeeServiceIntf employeeService;
    @Autowired
    public void setEmployeeService(EmployeeServiceIntf
        employeeService) {
        this.employeeService = employeeService;
    }
    @Test
    public void testEmployeeService_OneEmployee() {
        Employee empl = employeeService.getEmployee(4);
        String expected = "Amy Hilbert";
        String actual = empl.getFirstName() + "+" + empl.getLastName();
        Assert.assertEquals(expected, actual);
    }
}
```

Spring matches this name to the config file

256

Notes:

This code can be found in SpringTesting project in the workspace. To run this application, open the class in Eclipse, from the menu, select the **Run → Run As... Option** and choose **JUnit Application**.

That's it!

In the autowired setter above, note that Spring doesn't actually care what the setter method name is. It could be `setXYZ()` for example. However, it does look at the name of the variable passed into the method and compares it to the id of a configured bean.

The TestCase Explained

- ❖ `@RunWith()` → test should be performed using Spring's TestRunner
- ❖ `@ContextConfiguration()` → location of the Spring config file(s)
- ❖ `@Test` → defines the methods to test
- ❖ `@AutoWired` → perform any bean injections (based on property name and config file name matches)

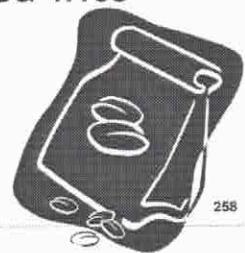
257

Notes:

Spring takes care of initializing the ApplicationContext and in this case even injects the employeeService object into the test case via the `@AutoWired` annotation.

Summary

- ❖ Spring combined with JUnit 4 makes unit testing services easy
- ❖ No base classes are required
- ❖ Annotations can be used to simplify test setup
- ❖ Beans can be automatically injected into test cases



Notes:

Exercise 3c: Spring and JUnit4

- ❖ Using JUnit4 and Spring annotations, create a TestCase to test the *Spring JDBC Product* code from exercise 3b.
- ❖ Use the source files provided in project **SpringLab03c** for this exercise.



259

Notes:

This page intentionally left blank!



260

260

Spring Framework(Part 1)

Appendices

Spring Framework Appendices

Table of Contents

Appendix A: HSQLDB	3
Appendix B: Web Tier Integration and JSF	7
Appendix C: Spring Remoting and Distributed Services	25
Appendix D: Spring and Web 2.0	39
Appendix E: Managing Applications Using JMX	59
Appendix F: Java Message Service	69

2

©2015 Computer Trilogy, Inc. ALL RIGHTS RESERVED

No part of this manual may be copied, photocopied, or reproduced in any form or by any means without permission in writing from the Author—Computer Trilogy, Inc. All other trademarks, service marks, products or services are trademarks or registered trademarks of their respective holders.

This course and all materials supplied to the student are designed to familiarize the student with the operation of the software programs. THERE ARE NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, MADE WITH RESPECT TO THESE MATERIALS OR ANY OTHER INFORMATION PROVIDED TO THE STUDENT. ANY SIMILARITIES BETWEEN FICTITIOUS COMPANIES, THEIR DOMAIN NAMES, OR PERSONS WITH REAL COMPANIES OR PERSONS IS PURELY COINCIDENTAL AND IS NOT INTENDED TO PROMOTE, ENDORSE, OR REFER TO SUCH EXISTING COMPANIES OR PERSONS.

Appendix A: HSQLDB

3

Notes on Using HSQL

- ❖ *HSQL DB* is an open source Java-based relational database used for this course
- ❖ It is easy to setup and work with
 - Consists of one Jar: **hsqldb.jar**
- ❖ To start the DB:
 - From within Eclipse
 - Expand the HSQLDB project and double-click **StartDB.bat**
- ❖ From a command-prompt:
 - From within the <workspace>/HSQLDB/data directory, type:
java -classpath <path_to_jar>/hsqldb.jar org.hsqldb.Server

4

Notes:

Notes on Using HSQL ...

- ❖ Stop the database using CTRL+C in the command window
- ❖ Start with a new, empty database:
 - ❑ Delete the contents of the data directory in the *HSQLDB* project after shutting down the server
- ❖ To bring up the *GUI DB Manager*
 - ❑ From within Eclipse
 - Expand the HSQLDB project and double-click **RunDbMgr.bat**
 - ❑ From a command-prompt:
 - From within the <workspace>/HSQLDB/data directory, type:
`java -classpath <path_to_jar>/hsqldb.jar
org.hsqldb.util.DatabaseManagerSwing`

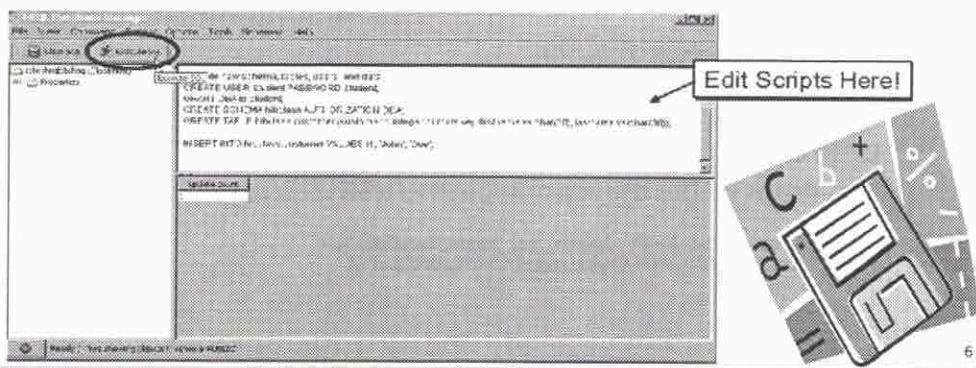
5

Notes:

When connecting to the DB using the GUI Database Mgr, select the type: "HSQL Database Engine Server". By default the user is SA and the password is left blank.

Notes on Using HSQL ...

- ❖ To run a script or query, start the GUI Database Mgr
 - Connect with type 'HSQL Database Engine Server'
 - Select **File → Open Script...** → (locate .script file)
 - Press **Execute SQL** button on the Toolbar



Notes:

Scripts may also be manually typed or even copied and pasted into the script editor and then executed.

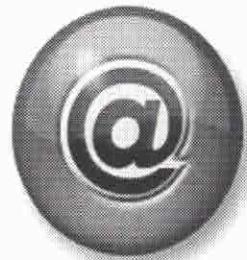
Appendix B: Web Tier Integration and JSF

7

7

Web Tier Integration and JSF Overview

- ❖ Spring-Enabling JSF-based Apps
- ❖ Spring to JSF Integration
- ❖ FacesContextUtils
- ❖ JSF to Spring Integration



8

Notes:

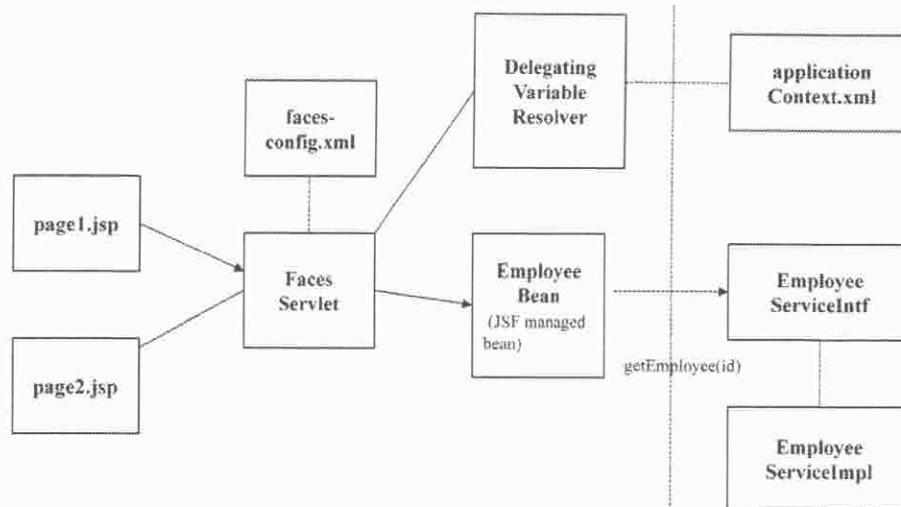
Spring JSF Support

- ❖ Spring and JSF have similar bean models
- ❖ Spring provides a VariableResolver class to help JSF beans access Spring beans
 - This results in a JSF container and a Spring IOC Container managing their own respective beans

9

Notes:

Our Example



10

Notes:

Setting Up web.xml

```
<web-app...>
    <listner>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listner>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>page1.faces</welcome-file>
    </welcome-file-list>
</web-app>
```

11

Notes:

As discussed earlier, integrate Spring by using a listener. The same is true for JSF apps. Set up the faces servlet as you normally would.

The Managed Bean: EmployeeBean

```
public class EmployeeBean {  
  
    private EmployeeServiceIntf employeeService;  
    private int id;  
    private Employee employee;  
  
    // getters and setters for these properties  
  
    public String find() {  
        employee = employeeService.getEmployee(id);  
        return "success";  
    }  
}
```

This is a Spring bean
injected by the JSF
container



12

Notes:

Notice the reference in the managed bean to the Spring-based EmployeeService. This will be initialized when Spring and JSF start up.

faces-config.xml

```
<faces-config>
    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>employeeBean</managed-bean-name>
        <managed-bean-class>
            com.company.springclass.jsf.beans.EmployeeBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>employeeService</property-name>
            <value>#{employeeService}</value>
        </managed-property>
    </managed-bean>
</faces-config>
```

This will cause JSF to locate beans in the Spring config file

13

Notes:

With the `<variable-resolve>` declaration, JSF will search its own config file for the desired bean first, then it will delegate requests to search the Spring config file.

applicationContext.xml

```
<beans ...>
    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close" >
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsq://localhost"/>
        <property name="username" value="student"/>
        <property name="password" value="student"/>
    </bean>
    <bean id="employeeService"
        class="com.company.springclass.services.EmployeeServiceImpl">
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
    </bean>
</beans>
```

14

Notes:

The Spring config file defines a bean called employeeService. This looks the same as it did in previous sections.

The JSPs

page1.jsp

```
<f:view>
  <h:form>
    <h:outputLabel for="id">Enter Employee ID: </h:outputLabel>
    <h:inputText id="id" value="#{employeeBean.id}" />
    <h:commandButton value="Find" action="#{employeeBean.find}" />
  </h:form>
</f:view>
```

page2.jsp

```
<f:view>
  <h1>Employee Info</h1>
  <ul>
    <li> FirstName: <h:outputText
        value="#{employeeBean.employee.firstName}" /> </li>
    <li> LastName: <h:outputText
        value="#{employeeBean.employee.lastName}" /> </li>
    <li> Dept Id: <h:outputText
        value="#{employeeBean.employee.deptId}" /> </li>
  </ul>
</f:view>
```

15

Notes:

The JSPs for our solution contain the input form and output pages respectively. Our employee data was stored as a reference in the EmployeeBean class, so it is easy to retrieve.

FacesContextUtils

- ❖ Another way to access Spring beans in JSF apps is to use *FacesContextUtils*
- ❖ A Spring-based **JSF Helper Utility**

```
EmployeeServiceIntf employeeService = (EmployeeServiceIntf)  
    FacesContextUtils.getRequiredWebApplicationContext(  
        FacesContext.getCurrentInstance()).  
        getBean("employeeService");
```



16

Notes:

JSF to Spring Injection

- ❖ The previous two examples inject Spring beans into a JSF web app
 - This results in two containers being used
- ❖ A cleaner approach is to inject JSF beans into Spring
 - This results in only a single (Spring) container managing beans

17

Notes:

Managing JSF Beans in Spring

❖ The Approach:

1. Set up a Spring *RequestContextListener*
2. Set up the *DelegatingVariableResolver* in *faces-config.xml*
3. Move the managed bean definition into the Spring config file
4. Remove the managed bean from *facesconfig.xml*



18

Notes:

web.xml

```
<web-app ...>  
  
    <listener>  
        <listener-class>  
            org.springframework.web.context.ContextLoaderListener  
        </listener-class>  
    </listener>  
  
    <listener>  
        <listener-class>  
            org.springframework.web.context.request.RequestContextListener  
        </listener-class>  
    </listener>  
  
    ... (same servlet and mapping for JSF as before) ...  
  
</web-app>
```

The *RequestContextListener* requires Spring 2.0 or later

19

Notes:

faces-config.xml

```
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>

  <navigation-rules>
    <from-view-id>/page1.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/page2.jsp</to-view-id>
    </navigation-rule>
  </navigation-rules>

</faces-config>
```

The *employeeBean* definition has
been removed from here



20

Notes:

applicationContext.xml

```
<beans...>
    <bean id="employeeBean"
          class="com.company.springclass.jsf.beans.EmployeeBean"
          scope="request">
        <property name="employeeService" ref="employeeService"/>
    </bean>

    <bean id="employeeService"
          class="com.company.springclass.services.EmployeeServiceImpl">
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
    </bean>
</beans>
```

JSF bean, *employeeBean*, now instantiated by Spring

21

Notes:

The *dataSource* bean definition was omitted for brevity.

Summary

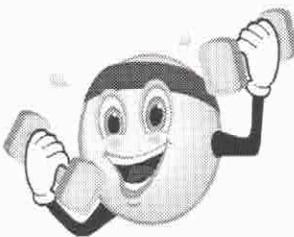
- ❖ Because both JSF and Spring have injection capabilities, there is an overlap of responsibilities
- ❖ There are 3 ways to integrate Spring and JSF applications:
 - ❑ Use a DelegatingVariableResolver for Spring to JSF integration
 - ❑ Use the FacesContextUtils class provided by Spring for Spring to JSF integration
 - ❑ Use a RequestContextListener for JSF to Spring integration

22

Notes:

Exercise 6

- ❖ Complete the **JSF-Spring** integration exercise detailed in *SpringLab06*
- ❖ Follow the additional instructions in *SpringLab06*



23

Notes:

This page intentionally left blank!



24

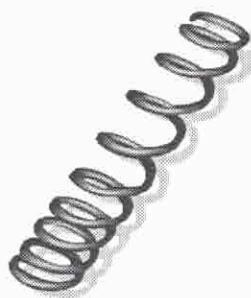
24

Appendix C: Spring Remoting and Distributed Services

25

Spring Remoting Overview

- ❖ Spring Remoting with RMI
- ❖ Remoting Configuration
- ❖ Remoting Example



26

Notes:

Remote Objects

- ❖ Historically, creating remotely accessed objects is a complex process
- ❖ Solutions require stubs and skeletons to be generated
- ❖ Client communicates with a stub
 - Client "appears" to be working with remote object



27

Notes:

Spring Remoting Services

- ❖ Spring supports distributed objects in several different ways:
 - RMI
 - JAX-RPC
 - JAX-WS
 - HTTP-Invoker



28

Notes:

There are several ways of remoting within Spring. While web services are commonly encountered, and Spring supports them with its JAX-WS SOAP capabilities, we will only tackle one form of remoting here: RMI.

Spring Remoting

- ❖ Spring Simplifies the remoting tasks by handling much of the "communications coding"
- ❖ Our example uses the DayOfWeek service
- ❖ DayOfWeek will be exposed by Spring
 - Spring will require some simple configuration

29

Notes:

Exposing the DayOfWeek Service

```
public interface DayOfWeek {  
    public String getDayOfWeek(int year, int month, int day);  
}  
  
-----  
  
public class DayOfWeekImpl implements DayOfWeek {  
    public String getDayOfWeek(int year, int month, int day) {  
        Calendar c = new GregorianCalendar(year, month, day);  
        return new SimpleDateFormat("EEEE").format(c.getTime());  
    }  
}
```

30

Notes:

The Remoting application begins with a service. Design a service based upon a clean, well-designed interface.

The Host Server

```
public class DayOfWeekHost {  
  
    public static void main (String[] args) throws Exception {  
  
        new FileSystemXmlApplicationContext (  
            "applicationContext.xml");  
        System.out.println("Server Started...");  
    }  
}
```

The server only needs to
instantiate the needed services

Because the RMIServiceExporter will be
started, the application remains alive

Running this Java class starts
the RMI application on the
server side

31

Notes:

A remoting application has two parts: the server and the client. These are separate (hence the name remote) applications running. The server is shown here. This code merely instantiates an IOC container. The IOC container then instantiates the needed services as defined in the config file.

applicationContext.xml

```
<beans ... >
    <bean id="dayOfWeekService"
          class="com.company.springclass.service.DayOfWeekImpl"/>

    <bean id="serviceExporter"
          class="org.springframework.remoting.rmi.RmiServiceExporter">
        <property name="serviceName">
            <value>DayOfWeek</value>
        </property>
        <property name="service">
            <ref local="dayOfWeekService"/>
        </property>
        <property name="serviceInterface">
            <value>com.company.springclass.service.DayOfWeek</value>
        </property>
        <property name="registryPort"><value>1099</value>
        </property>
        <property name="servicePort"><value>9226</value>
        </property>
    </bean>
</beans>
```

32

Notes:

The RmiServiceExporter's job is to create and register service properties for an RMI registry. The typical port for an RMI registry is 1099. This is the default if not specified. The service port is the port number that the RMI service will communicate on. The serviceName will be the name of the RMI registry entry for this service. The service property is very important as it defines the Spring bean to use for the RMI service.

The RMI Client

- ❖ The client application is independent of the server
 - It will have its own Spring configuration and IOC container
 - The client must be given the service interface (often termed the remote interface)
 - This is the DayOfWeek class in our example

33

Notes:

The Client Code

```
public class DayOfWeekClient {  
    private DayOfWeek dayOfWeekService;  
    public void setDayOfWeekService(DayOfWeek dayOfWeekService){  
        this.dayOfWeekService = dayOfWeekService;  
    }  
    public static void main(String[] args) throws Exception {  
        ApplicationContext ctx =  
            new FileSystemXmlApplicationContext(  
                "applicationContext-client.xml");  
        DayOfWeekClient dayOfWeekClient =  
            (DayOfWeekClient) ctx.getBean("dayOfWeekClient");  
        dayOfWeekClient.run();  
    }  
    public void run() {  
        System.out.println(dayOfWeekService.getDayOfWeek(2009, 0, 1));  
    }  
}
```

34

Notes:

This is the client code for our client-server application. Notice it has a reference and setter method to the service interface. In the *main()* method, once the app starts, the client's Spring IOC container is created. This causes the Spring config file to be read and the appropriate beans to be instantiated. As you can see on the next slide, the *dayOfWeekService* is injected by Spring into this application. Therefore, we instantiate the client and have at our disposal the *dayOfWeekService* as a result.

applicationContext-client.xml

```
<beans ... >
    <bean id="dayOfWeekService"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl">
            <value>rmi://localhost:1099/DayOfWeek</value>
        </property>
        <property name="serviceInterface">
            <value>com.company.springclass.service.DayOfWeek</value>
        </property>
    </bean>

    <bean id="dayOfWeekClient"
        class="com.company.springclass.remoting.client.DayOfWeekClient">
        <property name="dayOfWeekService">
            <ref local="dayOfWeekService"/>
        </property>
    </bean>
</beans>
```

The protocol, host, port, and serviceName of the Server's RMI Service

Client instance is injected with the dayOfWeekService

35

Notes:

The RmiProxyFactorybean has the task of making a remote object "appear" to be local to a client. It does this by generating a proxy component that implements the same interface as the service.

Summary

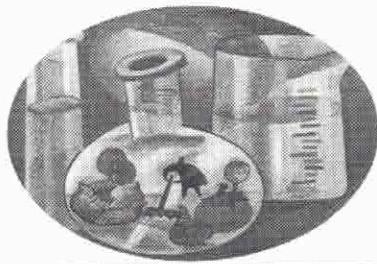
- ❖ Spring Remoting makes client / server development an easy task by removing much of the ugly coding of the proxy code on the server.
- ❖ Create separate Spring configurations for both the server and the client
- ❖ The client will need to configure an RmiProxyFactoryBean
- ❖ The server will need to define a ServiceExporter

36

Notes:

Lab 7: Spring Remoting

- ❖ Complete the Employee Service application following the additional instructions in **SpringLab07a** and **SpringLab07b**
 - ❖ **SpringLab07a** is a Host Application
 - ❖ **SpringLab07b** is a Client Project

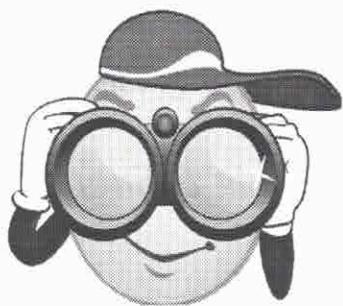


Start with Lab 07a...

37

Notes:

This page intentionally left blank!



38

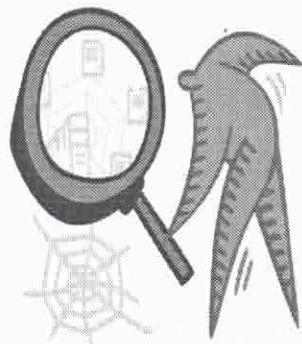
Appendix D: Spring and Web 2.0

39

39

Spring and Web 2.0 Overview

- ❖ What is DWR?
- ❖ Spring DWR Integration



40

Notes:

What is DWR?

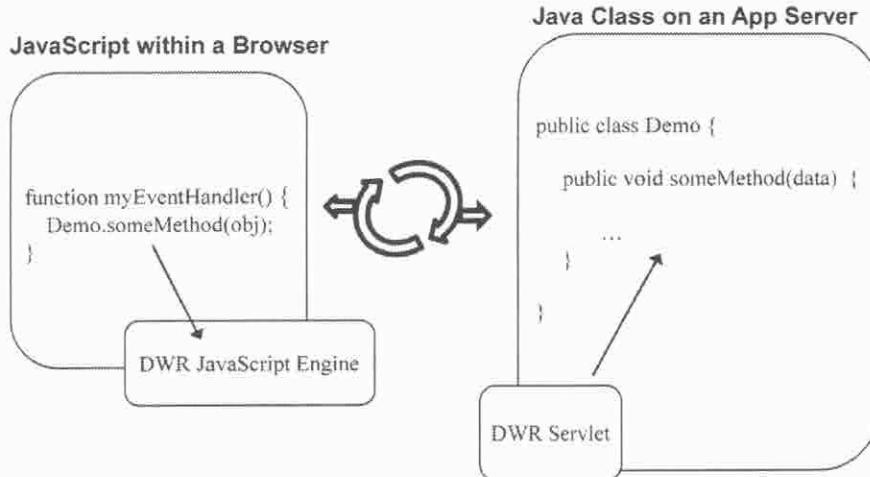
- ❖ Direct Web Remoting (DWR)
 - A framework for making Java calls from JavaScript
 - Passing objects back and forth from JavaScript to a Java-based App Server
 - Automatic JavaScript code generation
 - Makes the browser seem like a remote client to the server

41

Notes:

DWR is a Client-Server solution in which a JavaScript code can invoke Java functions very easily without creating a lot of 'hookup' code. In fact, DWR will manage and create much of the needed communications code for you.

How Does it Work?



42

Notes:

A developer writes a JavaScript function that appears to directly invoke a Java class and function on the server-side. In reality, the DWR JavaScript engine intercepts this function call, handles the Ajax communications to the DWR servlet, which invokes the method in the Java class. The DWR framework then passes the response back to the JavaScript Engine and invokes a developer-created callback in the JavaScript code.

Setting Up DWR

- ❖ Down the JAR

<http://directwebremoting.org/>

- ❖ Drop *dwr.jar* (and commons-logging.jar) into the WEB_INF/lib directory



43

Notes:

To begin using DWR in an application, simply download the JAR from the provided web site and place it in the **lib** directory of your web application, along with Jakarta commons logging JAR.

Setting Up DWR ...

- ❖ Establish the DWR Servlet in web.xml:

```
<servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>
        org.directwebremoting.servlet.DwrServlet
    </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

To see DWR debug info, visit
local:8080/SpringDWR01/dwr/index.html

44

Notes:

Add the DWR servlet into your web.xml along with your desired mapping. The debug init-param allows for directory browsing of all configured DWR objects that are exposed.

Setting Up DWR ...

- ❖ Set up the meta-data file *dwr.xml*:

```
<!DOCTYPE dwr PUBLIC  
  "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"  
  "http://getahead.org/dwr/dwr20.dtd">  
<dwr>  
  <allow>  
    <create creator="new" javascript="HelloWorld">  
      <param name="class"  
             value="com.company.springclass.dwr.beans.HelloWorld"/>  
    </create>  
  </allow>  
</dwr>
```

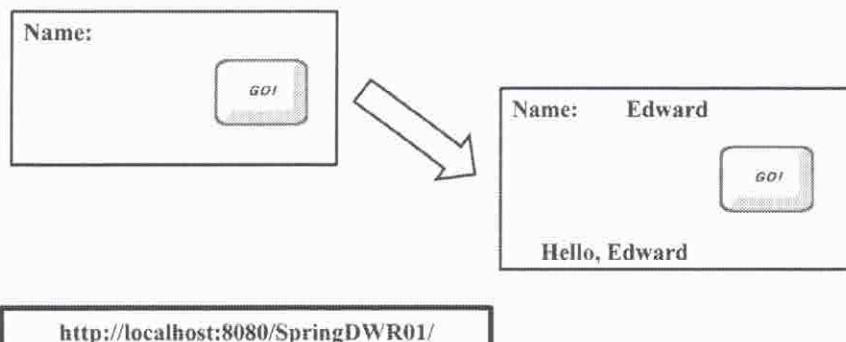
45

Notes:

Create the *dwr.xml* metadata file that the DWR servlet reads. This file is placed in the WEB-INF directory and should be called *dwr.xml*. This file will tell DWR what calls to allow on certain classes as well as what JavaScript code to create. That's it for the setup.

Simple DWR Example

- ❖ The following example illustrates how to make a DWR call



46

Notes:

In this example, we will create a form that submits data and gets a response from the server as shown using DWR. The URL to view this example (server must be running) is shown in the slide.

Create the Java Class

Step 1

```
package com.company.springclass.dwr.beans;

public class HelloWorld {
    public String greeting(String name) {
        if (name == null) name = "friend";

        return "Hello, " + name;
    }
}
```

47

Notes:

This will be our class that is invoked on the server-side. We will invoke it from our JavaScript code. The method *greeting()* accepts a value and returns a value back to the client.

Create the dwr.xml Entry

Step 2

```
<!DOCTYPE dwr PUBLIC  
        "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"  
        "http://getahead.org/dwr/dwr20.dtd">  
  
<dwr>  
    <allow>  
        <create creator="new" javascript="HelloWorld">  
            <param name="class"  
                  value="com.company.springclass.dwr.beans.HelloWorld"/>  
        </create>  
    </allow>  
</dwr>
```

The name of the automatically generated JavaScript file. A script tag will be needed in the HTML for this file

The name of the Java class that will be instantiated and accessed at the method level

48

Notes:

This file identifies how DWR should create the JavaScript files and what files to allow access to.

The create element tells DWR that a server-side class should be exposed to Ajax requests and defines how DWR should obtain an instance of that class to remote. The creator attribute here is set to the value new, meaning that DWR should call the class's default constructor to obtain an instance. Other possibilities are to create an instance through a fragment of script using the Bean Scripting Framework (BSF), or to obtain an instance via integration with the IOC container, Spring. By default, when an Ajax request to DWR invokes a creator, the instantiated object is placed in page scope and therefore is no longer available after the request completes.

Include <script> Tags

Step 3

- Be sure to put script tags in the HTML

```
<script type="text/javascript"  
       src="/SpringDWR01/dwr/interface/HelloWorld.js"> </script>  
<script type="text/javascript"  
       src="/SpringDWR01/dwr/engine.js">  
  </script>
```

Matches javascript attribute in
the dwr.xml file

Contains the main
DWR functions

49

Notes:

You do not create HelloWorld.js. This file is generated by DWR based on your value in the dwr.xml file shown on the previous slide. This file will be referenced from the /<APPNAME>/dwr/interface directory structure as shown.

Create the HTML Page

Step 4

- The form invokes a JavaScript function:

```
<div id="wrapper">
  <form onsubmit="return handleForm();">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name"></input>
    <input id="submit" type="submit" value="Go!"/> </input>
  </form>
  <div id="resultsDiv"></div>
</div>
```

50

Notes:

Our HTML form invokes the *handleForm* JavaScript function.

The JavaScript (the Client)

Step 5

- The JavaScript invokes the DWR JavaScript engine (which in turn invokes the servlet)

```
var handleForm = function() {  
    var input = document.getElementById('name');  
    HelloWorld.greeting(input.value,  
        { callback: myCallback,  
          timeout:5000,  
          errorHandler:function(message) {  
              alert("Oops: " + message); }  
    });  
    return false;  
}  
  
var myCallback = function(results) {  
    document.getElementById('resultsDiv').innerHTML = results;  
}
```

Get the input

Name of callback

Java code to invoke

Callback invoked by DWR containing return value

51

Notes:

handleForm() is called when the form is submitted. The input value is retrieved and passed to the Java method by writing JavaScript code. The JavaScript code actually invokes the DWR JavaScript engine, which in turn talks to the DWR servlet. The response is sent back to the page and your callback function (in this case, called *myCallback*) is invoked.

Integrating Spring and DWR

- ❖ DWR 2.0 allows for Spring to maintain all necessary DWR config info
 - *dwr.xml* can be removed
- ❖ DWR can invoke Spring services directly
 - Load a different DwrServlet
 - This one is Spring-aware
 - `org.directwebremoting.spring.DwrSpringServlet`

52

Notes:

web.xml

```
<servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>
        org.directwebremoting.spring.DwrSpringServlet
    </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>allowsScriptTagRemoting</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

A Spring-aware
DWR Servlet

53

Notes:

applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.directwebremoting.org/schema/spring-dwr
                           http://www.directwebremoting.org/schema/spring-dwr-3.0.xsd">
    ...
</beans>
```

DWR-specific tags will be used,
so a new namespace is added to
the Spring config file

54

Notes:

The contents of the config file are on the next slide.

applicationContext.xml ...

```
<bean id="employeeService"
      class="com.company.springclass.services.EmployeeServiceImpl">
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
    <dwr:remote javascript="EmployeeService" />
</bean>

<dwr:configuration>
  <dwr:convert type="bean"
    class="com.company.springclass.beans.Employee" />
</dwr:configuration>

</beans>
```

Allow DWR to create JavaScript for this service

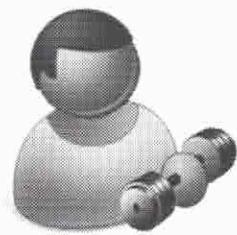
Define how to use the data returned from the service, in this case as properties from an Employee object

55

Notes:

Lab 8: Spring and DWR

- ❖ Complete the application following the additional instructions in **SpringLab08**
- ❖ This exercise will use Spring and DWR and make use of the *EmployeeService*



56

Notes:

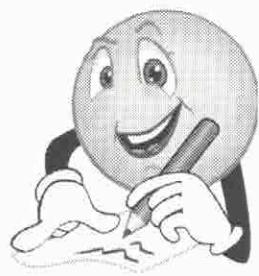
Summary

- ❖ DWR can make remote calls to a Spring-based service easy
- ❖ Ajax-based Spring service solutions bring powerful new Web 2.0 capabilities to web applications
- ❖ When a service returns a business object, a converter configuration needs to be set up

57

Notes:

This page intentionally left blank!



58

Appendix E: Managing Applications Using JMX

59

Overview

- ❖ Introducing JMX
 - JMX Basic Definition
 - MBean Registration
 - Spring and JMX



60

Notes:

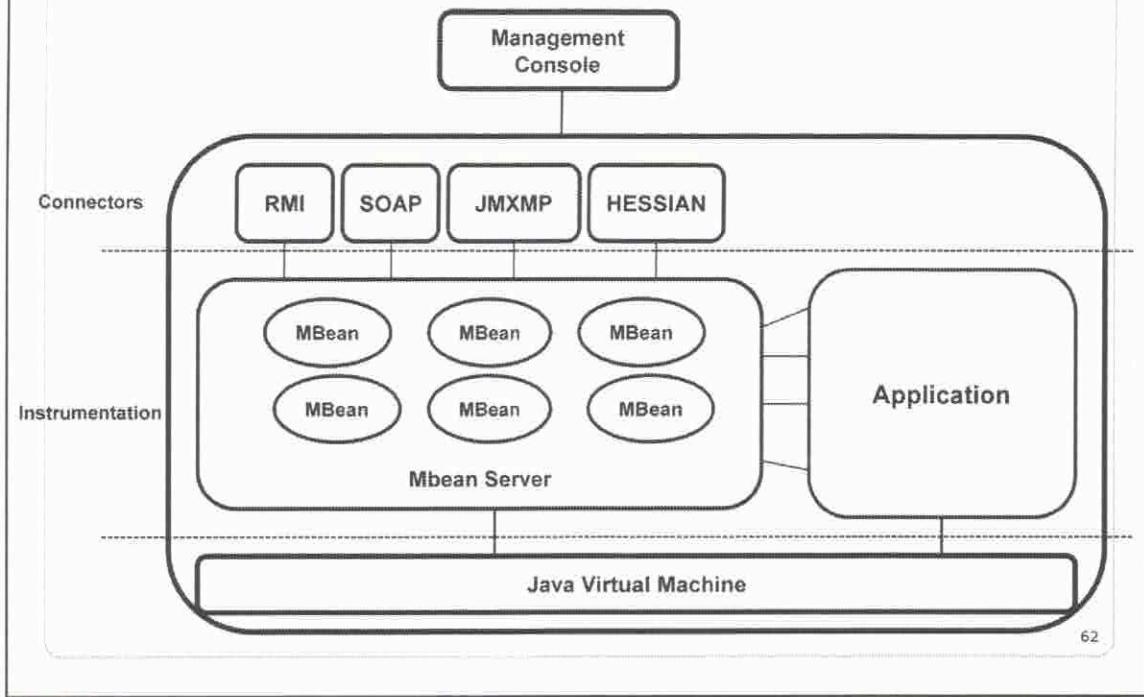
JMX

- ❖ JMX not JMS
- ❖ Java Management eXtension
- ❖ Mature Standard (JSR-3)
- ❖ Provides Managed Bean (MBean) Server
 - Bean registration / discovery / query
 - Exposes beans information
- ❖ Built-in Event / Notification Model
- ❖ Built-in Monitoring / Timer Services
- ❖ Part of Java 5

61

Notes:

JMX Management Architecture



Notes:

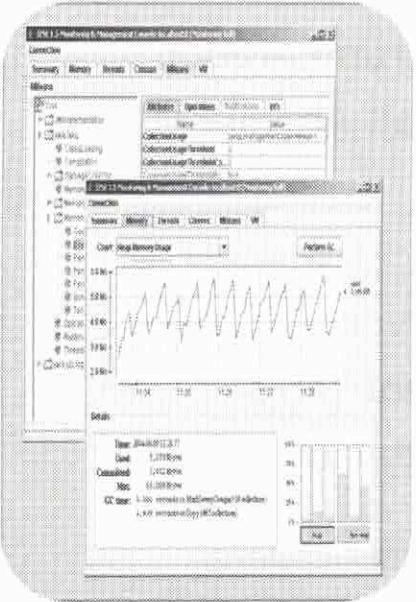
JMX Managed Beans (MBeans)

- ❖ Standardized API for Instrumentation
- ❖ Exposes Properties (MBean Attributes)
- ❖ Exposes Operations
- ❖ Emit Event Notifications
- ❖ Hosted within MBean Server
- ❖ Accessible via Management Tools
 - Query MBean Server for attribute values
 - Access MBean operations
 - Register for Mbean event notifications

63

Notes:

JConsole



- ❖ Desktop Management Console
- ❖ Graphically Expose MBeans
- ❖ Part of Java 5
- ❖ Info on GC, Memory, Threads
- ❖ Support Remote Connection
- ❖ Built-in Security
- ❖ Requires VM-Argument Switch

64

Notes:

Spring and JMX

- ❖ Spring makes JMX development easier
- ❖ MBean Exporter Component
 - Generates/registers dynamic MBean component
- ❖ Exposes regular POJO as managed objects
 - No special interfaces or naming patterns required
- ❖ Declarative MBean Registration
- ❖ Flexible Management API
 - Java annotations / Common Attributes
 - Expose via interfaces, method names, POJO's

65

Notes:

JMX Usage Pattern

- ❖ JVM Profiling, Monitoring & Control
- ❖ Application Monitoring
 - Exposing instrumented aspects
 - Monitoring of activities and dynamic values
 - Notifying or reacting to state changes
- ❖ Application Control
 - Control life cycle of application components
 - Expose settings as push-button controls
- ❖ Configuration
 - Runtime update of application state



66

Notes:

Summary

- ❖ Spring, Spring AOP, and JMX (included in Java 5) are potent combinations for creating highly manageable applications.
- ❖ The Spring Framework provides a set of API that makes it easy to develop manageable applications using a declarative XML configuration file to expose POJO as managed beans.
- ❖ JMX and related tools can be used to provide runtime application monitoring, control, and configuration services.
- ❖ Most common case is application monitoring. Use Spring AOP to cut through your application execution sequence to collect data and push that data onto dedicated managed beans.
- ❖ Use dedicated POJOs to expose management information and controls. This allows you to separate manageability from the core business logic of your application.

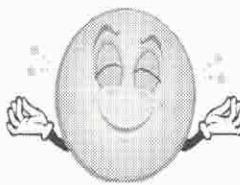
67

Notes:

Code Samples

- ❖ JVM command line parameter for jConsole

-Dcom.sun.management.jmxremote



68

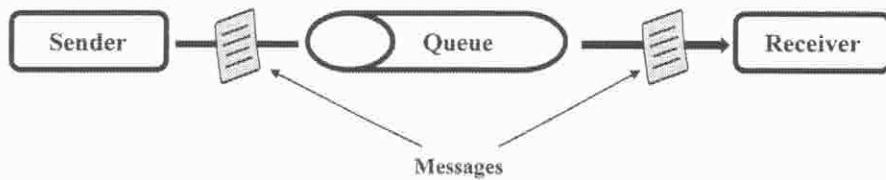
Notes:

Appendix F: Java Message Service

69

JMS Overview

- ❖ Asynchronous Messaging
 - Brokers and Destinations
- ❖ Queues → One Sender / One Receiver
 - Message is removed from queue once received
 - Multiple receivers can listen to the same queue



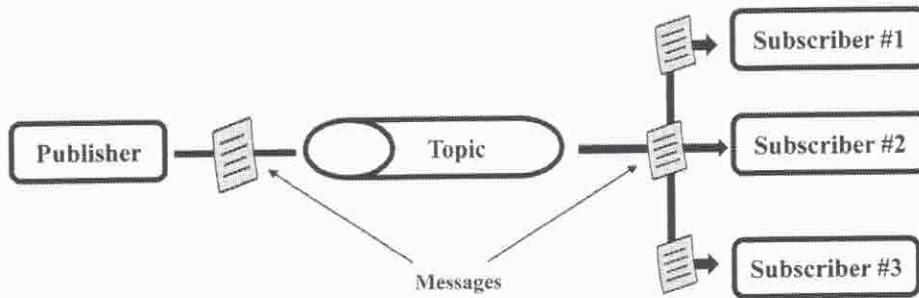
70

Notes:

Topics

- ❖ Publish / Subscribe

- All receivers receive a copy of the message



71

Notes:

Benefits

- ❖ No waiting
- ❖ Service specifics are not required
- ❖ Location independence
- ❖ Guaranteed delivery



72

Notes:

Steps

- ❖ Set-up a Message broker → *ActiveMQ*
- ❖ Add ***activemq.jar*** to the classpath
- ❖ Run ***activemq.bat***
- ❖ Create a connection factory

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

73

Notes:

Steps ...

- ❖ Declare a message destination

- Queue

```
<bean id="rantzDestination"
      class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg index="0" value="rantz.marketing.queue"/>
</bean>
```

- Topic

```
<bean id="rantzDestination"
      class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg index="0" value="rantz.marketing.topic"/>
</bean>
```

74

Notes:

Working with JMS

- ❖ JMS code is similar to JDBC
- ❖ JMS Template
 - ❑ Connection
 - ❑ Obtain a session
 - ❑ Send/receive messages
 - ❑ JMSEException handling
 - ❑ Configuration

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

75

Notes:

Sending Messages

- ❖ Sending a message, the Big Picture



76

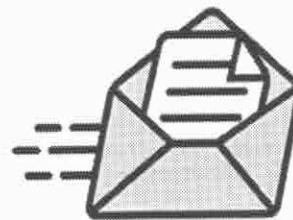
Notes:

Sending Messages ...

❖ The Code

```
package com.roadrantz.marketing;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import com.roadrantz.domain.Mortorist;

public class RantzMarketingGatewayImpl
    implements RantzMarketingGateway {
    public RantzMarketingGatewayImpl () {}
```



77

Notes:

Sending Messages ...

```
public void sendMotoristInfo(final Motorist motorist) {
    jmsTemplate.send (                                     Sends Message
        destination,                                ←————— Specifies destination
        new MessageCreator() {
            public Message createMessage(Session session)
                throws JMSException {
                    MapMessage message = session.createMapMessage();
                    message.setString("lastName", motorist.getLastName());
                    message.setString("firstName", motorist.getFirstName());
                    message.setString("email", motorist.getEmail());

                    return message;
                }
    });
}

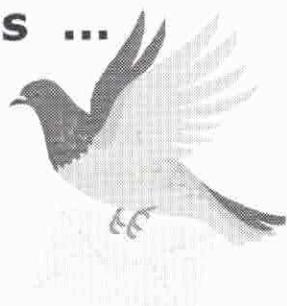
private JmsTemplate jmsTemplate;
public void setJmsTemplate(JmsTemplate jmsTemplate) {   Injects
    this.jmsTemplate = jmsTemplate;                      JmsTemplate
}
}

private Destination destination;
public void setDestination(Destination destination) {   Injects
    this.destination = destination;                     Destination
}
}
```

78

Notes:

Sending Messages ...



❖ Additional Wiring

```
<bean id="marketingGateway"
      class="com.rodrantz.marketing.RantzMarketingGatewayImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
    <property name="destination" ref="rantzDestination" />
</bean>
```

79

Notes:

Consuming a Message

- ❖ Receiving messages with JmsTemplate



80

Notes:

Consuming a Message ...

❖ The Code

```
package com.roaddrantz.marketing;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.JmsUtils;

public class MarketingReceiverGatewayImpl {
    public MarketingReceiverGatewayImpl() {}

    public SpammedMotorist receiveSpammedMotorist() {
        MapMessage message = (MapMessage) jmsTemplate.receive(); ← Receives
                                                                message

        SpammedMotorist motorist = new SpammedMotorist();
        try {
            motorist.setFirstName (message.getString("firstName"));
            motorist.setLastName (message.getString("lastName"));
            motorist.setEmail (message.getString ("email"));
        } Creates
              object from
              message
    }
}
```

81

Notes:

Consuming a Message ...

❖ The Code (cont'd)

```
    } catch (JMSEException e) {
        throw JmsUtils.convertJmsAccessException (e); ←———— Converts any
    } JMSEException
    return motorist;
}

//injected
private JmsTemplate jmsTemplate;
public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
}
```

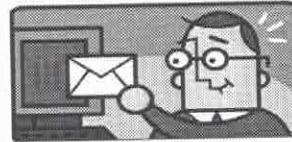
82

Notes:

Consuming a Message ...

❖ Configuration

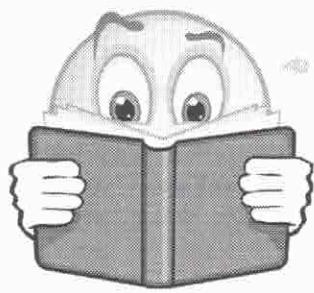
```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="rantzDestination" />
</bean>
```



83

Notes:

This page intentionally left blank!



84

84