Spring Framework
Table of Contents

2

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:

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.

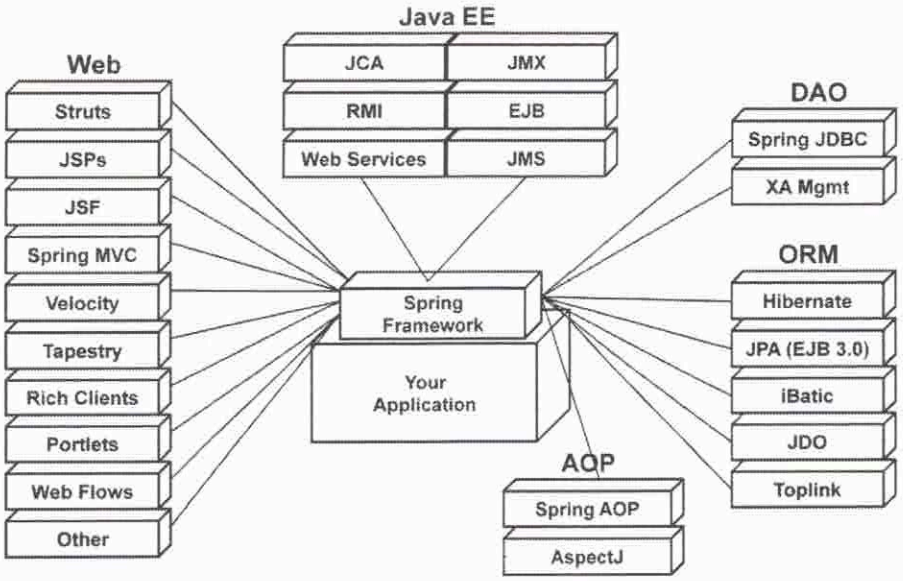
State of Spring

- 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:

Spring and Integration



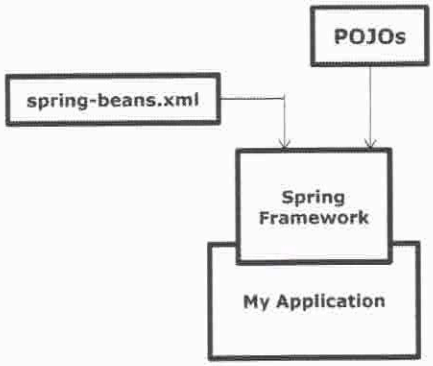
10

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:

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);
    }
}
```

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

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

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 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.

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.

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 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;
```

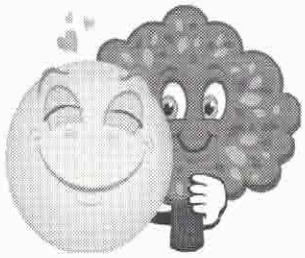
Summary

- 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:

This page intentionally left blank!



Spring IOC Overview

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

28

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 ...

```
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:

Constructor Injection

applicationContext-beans.xml

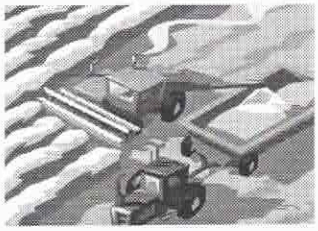
```
<?xml version="1.0" encoding="UTF-8"?>

<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:

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 ...

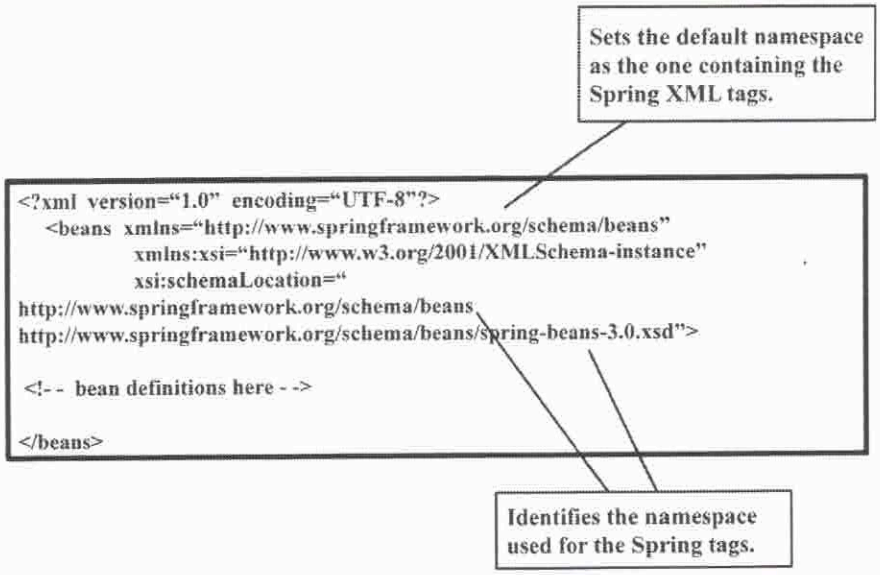
```
public class Driver
{
        public static void main (String[] args)
        {
                ApplicationContext context =
                    new ClassPathXmlApplicationContext(
                                        "applicationContext-beans.xml");
                Order order = (Order) context.getBean("orderBean");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

Outputs: 100 John Smith 1 0.0

38

Notes:

Understanding the Schema Attributes



40

Notes:

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 />
      <contructor-arg />
</bean>
```

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

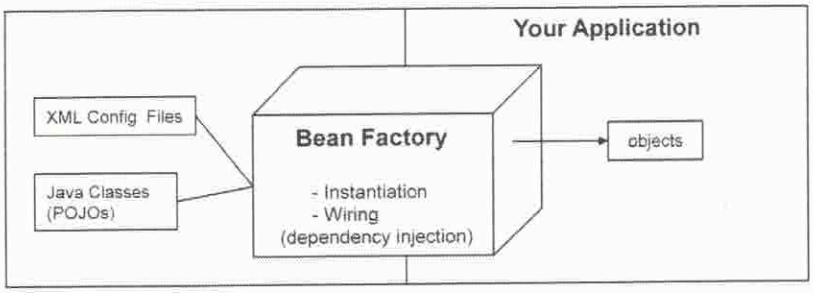
42

Notes:

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

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.

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 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 ...

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

Configures beans from multiple files

applicationContext-dao.xml

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

applicationContext-services.xml

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

50

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:

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>
```

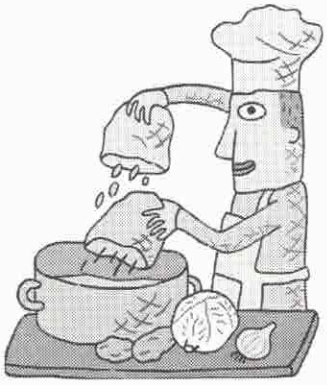
Passes a list of product beans (only one) into the constructor of the CatalogManagerImplementation class

54

Notes:

Work with the Beans

```
CatalogManager catalog =
                    (CatalogManager) context.getBean("catalog");

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



56

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"/>
```

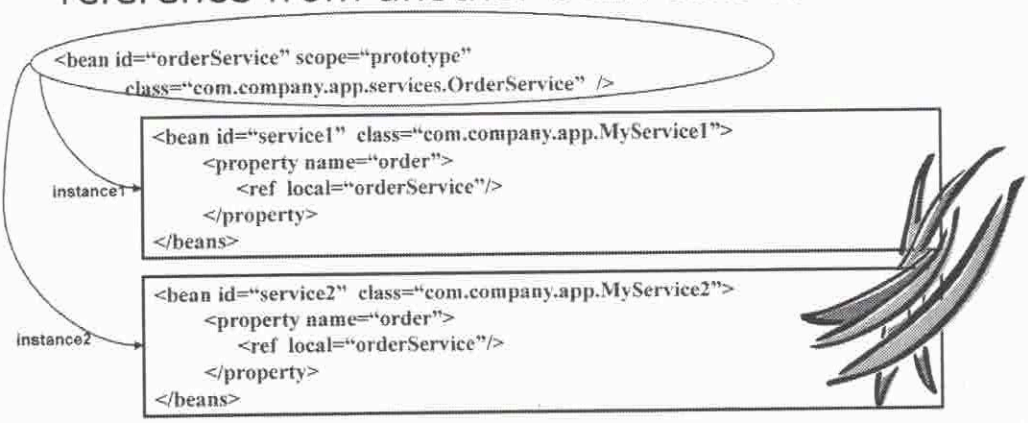
58

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.

Prototype Beans

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



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.

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.

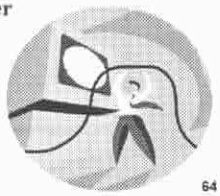
Spring-WebApp Configuration ...

```
<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)



64

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>
```

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.

Autowiring

- *Autowiring* within Spring is a means for the container to try to figure out dependencies automatically
- While this feature is powerful it <u>should generally not</u> 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 ...

- 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) {
            this.dao = dao;

        }
}
```

Take note of the names used

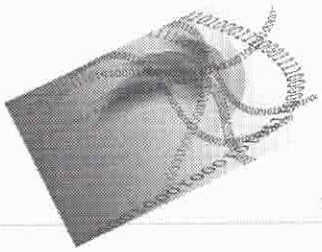
70

Notes:

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

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.

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:

Design Considerations: Issues with Annotations

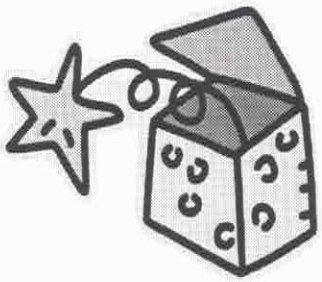
- Arguments *against* the use of annotations in Spring-based apps:
  - These wirings are configuration metadata that should be kept separate from the source code
  - They depend on proper variable naming conventions, which could change if a team member is not careful

76

Notes:

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: