

# Introduction to **spring**

**Concepts, Technical Walkthrough,  
& Application Development**

**Covering Core Spring 4.3.1 RELEASE**



# Presentation Revision History



Date	Version	Comments
22/07/2014	1.0	Covering Spring 3.2.11 Release
14/04/2016	2.0	47 Lab Exercises included.
09/08/2016	3.0	Updated the complete presentation to cover Spring 4.3.1 Release.

**Presentation Revision – v3.0**

**Last Updated on: 11<sup>st</sup> August 2016**

**Prepared by: Vijaya Raghava Vuligundam**

**Email: vvijayaraghava@hotmail.com**

**Mobile: +91.900.076.7644**

**Note: Special Thanks to Kaushik from JavaBrains, as a few concepts were covered with his examples for easier end-user understanding.**





Attendees are expected to have an understanding of the following:

- XML Concepts
- Experience with developing applications using Java / J2EE Technologies.
- Usage of tools including RAD and/or Eclipse IDE
- Design Patterns including the following:
  - Factory Method
  - Abstract Factory
  - Service Locator
  - Model View Controller





Introduce yourself and provide your:

- Name and Project to which you are working for
- Current Job Roles & Responsibilities, Total Experience
- Experience with Java/J2EE and XML Technologies
- Expectations from this training
- Goals you hope to achieve from this training





- This course is designed for Associates, who had experience with developing applications using Java / Java EE Technologies and involved in the exchange of information using XML as the data transport mechanism.
- 50% Theory - 50% Lab Practices
- 7 Self Study Notes
- 47 Lab Practices including 3 self study exercises.
- 74 Quiz Questions.



# Table of Contents



- 1 Problem with Dependencies
- 2 Spring: What's it all about ?
- 3 Setting up the Development Environment
- 4 Spring Framework Fundamentals
- 5 Spring Bean Concepts
- 6 Dependency Injection
- 7 Advanced Spring Bean Concepts

# Table of Contents



- |    |  |
|----|--|
| 8  | Annotation-based Container Configuration |
| 9  | Aspect Oriented Programming              |
| 10 | Best Practices                           |



# Agenda



- ▶ Problem with Dependencies
- ▶ Spring: What's it all about ?
- ▶ Setting-up the Development Environment
- ▶ Fundamentals of Spring Framework
- ▶ Spring Bean Concepts



Day #1



# Agenda



- ▶ Dependency Injection
- ▶ Spring Configurations



day #2



# Agenda



- ▶ Advanced Spring Bean Concepts
- ▶ Annotations



day #3



# Agenda



- ▶ Aspect Oriented Programming
- ▶ Best Practices



Day #4

# Expectations upon the completion of the course



- Understand the basic concepts of Spring.
- Understand the roles and responsibilities of Spring IOC Container
- Dependency Injection & Spring Bean Concepts
- Various Spring Configurations & Annotations
- Aspect Oriented Programming concepts
- Best Practices
- All set for working on your project assignment !!!



For the complete source code of 47 Lab Exercises, please drop me an email to: [vvijayaraghava@hotmail.com](mailto:vvijayaraghava@hotmail.com). I will reply back at the earliest possible time.

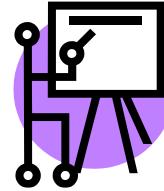




## Unit 1



### Problem with Dependencies



## Agenda



Problem with dependencies



An Example

## – Drawing Application

Circle

draw()

Triangle

draw()

Application Class

```
Circle circle = new Circle()  
circle.draw();
```

```
Triangle triangle = new Triangle()  
triangle.draw();
```

# High Dependency = High Responsibility 😞



```
public class Triangle {  
    public void draw() {  
        System.out.println(" Triangle Drawn ");  
    }  
}
```

```
public class DrawingApp {  
    public static void main(final String[] args)  
    {  
        Triangle triangle = new Triangle();  
        triangle.draw();  
    }  
}
```

## Requirement:

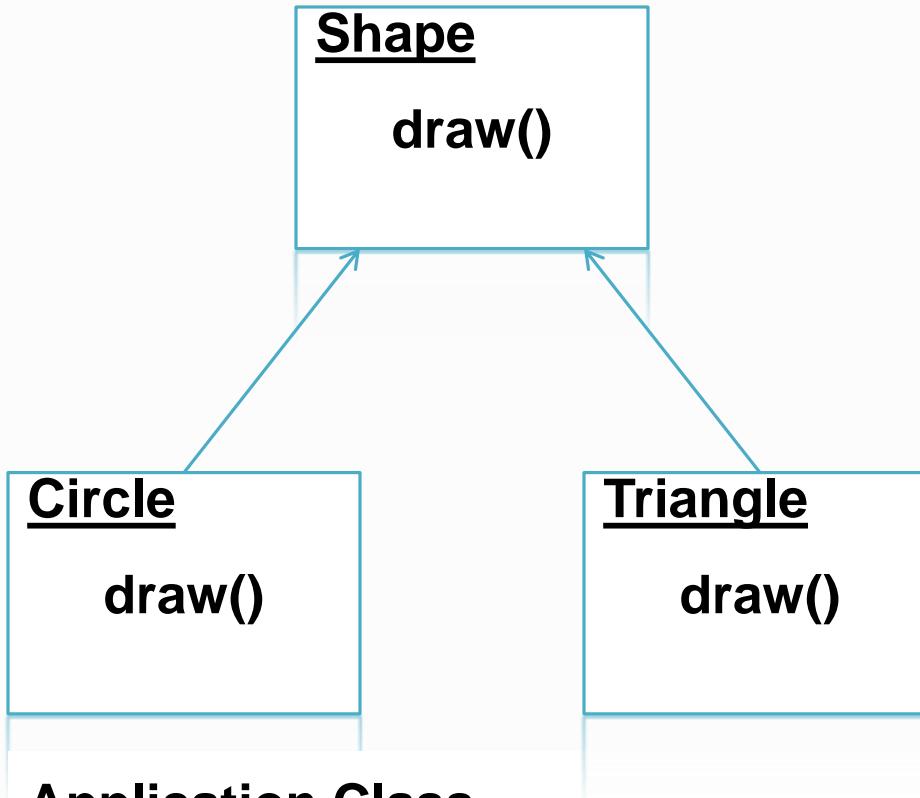
Change Triangle to Circle.



## **Too many changes.**

1. Write a standalone Circle class
2. Change the object declaration to Circle
3. Change the reference
4. Compile again
5. Test

# A bit less dependency ☺ – Using Polymorphism



## Application Class

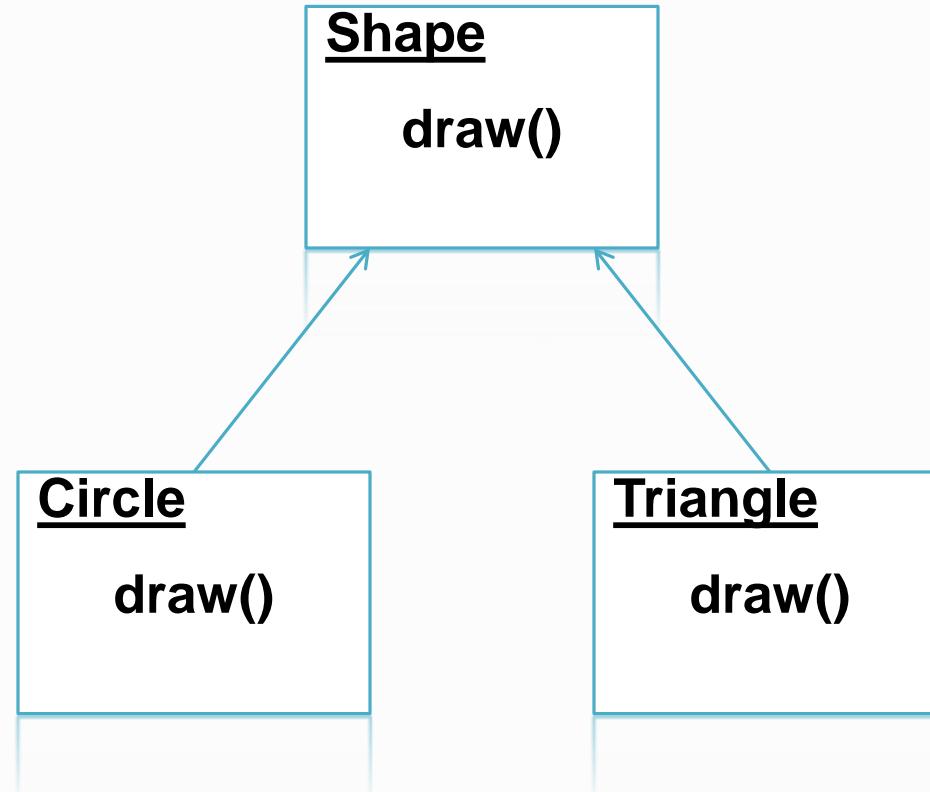
```
Shape shape = new Circle()
shape.draw();
```

```
Shape shape = new Triangle()
shape.draw();
```

**Requirement:** Change Triangle to Pentagon !!!

Again some changes.

1. Pentagon implements Shape
2. Change the object declaration to Circle
3. **~~Change the reference~~**
4. Compile again
5. Test



## Application Class

```
public void drawMethod(Shape shape) {  
  
    shape.draw();  
}
```

## Somewhere else in the class

```
Shape shape = new Circle();  
drawMethod(shape);
```

## Application Class

**Circle**

**draw()**

## Application Class

```
public void drawMethod(Shape shape) {  
    shape.draw();  
}
```

## Somewhere else in the class

```
Shape shape = new Circle();  
drawMethod(shape);
```



---

**GOT QUESTIONS?**

---

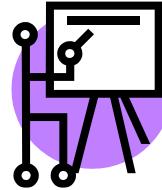




## Unit 2



# Spring: What's it all about ?



## Agenda

- 🎯 Complexity of Enterprise Application Development
- 🎯 History of Spring
- 🎯 Spring – Evolution over Intelligent Design
- 🎯 The Dependency Inversion Principle
- 🎯 Spring – Features, Benefits
- 🎯 Checkpoint Summary
- 🎯 Quiz

- **J2EE Specification promised Scalability, Security and High Availability.**
  - Enterprise JavaBeans (EJB), as part of the J2EE suite of specifications from Sun, were intended to be as reusable and portable as their non-enterprise counterparts, plain JavaBeans.
- **Problem with the Enterprise Java Beans (EJB)**
  - Many interfaces and configuration files required to create an EJB were awkward, tedious, and prone to error.
  - The EJBs' marriage to the container made unit testing close to impossible.
  - Applications became heavy in the bulk of extra container features.



## Solution:

EJB 3.0 has come up to address some of these issues. But it's too little and too late !!!

# What you need is

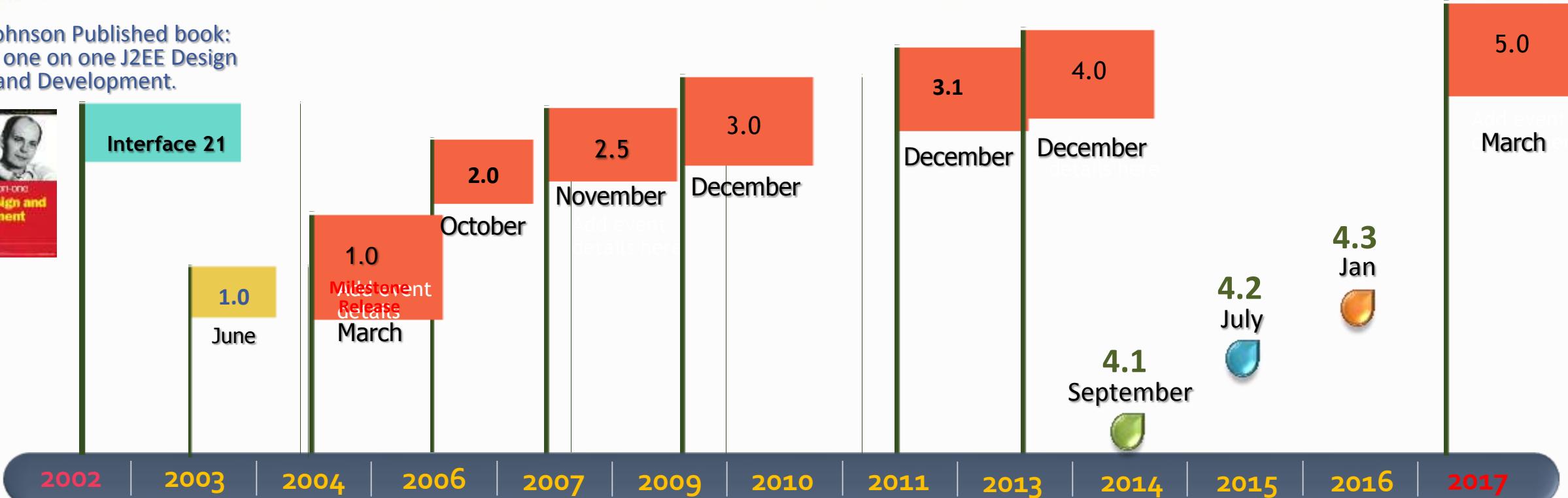


- Your components should run with or without a container.
- Be testable with minimal to no intrusion from outside classes.
- Components should have a life outside of the framework.
- A framework to help you put into place some established best practices for your applications

# A little *History* does make sense



Rod Johnson Published book:  
Expert one on one J2EE Design  
and Development.



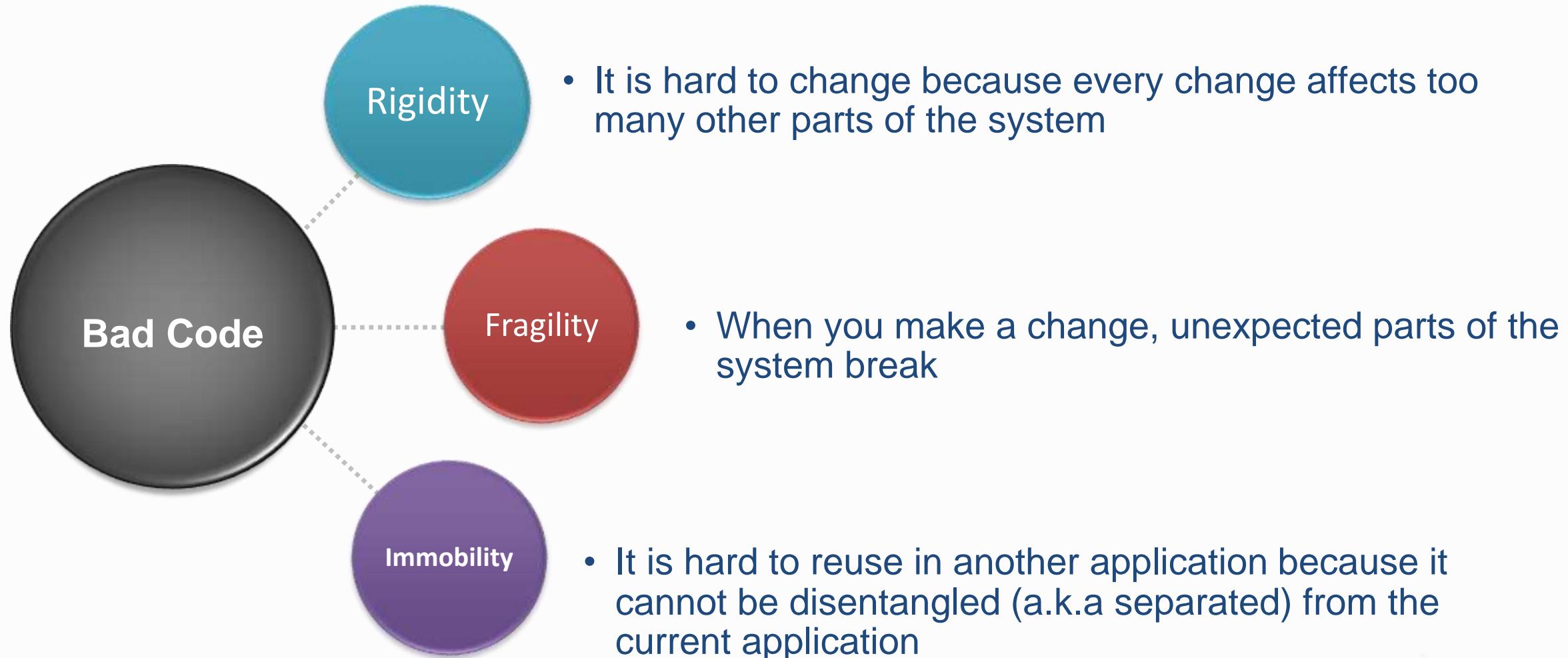
# Spring, Java and Java EE compatibility



Spring	Java Version	Java EE Version
1.x	Full Support for JDK 1.3, 1.4	Java EE 1.3, 1.4 are fully supported.
2.0.x	Full Support for JDK 1.3, 1.4 and 1.5	Java EE 1.3, 1.4 are fully supported.
2.5	Full support for Java SE 1.4.2 or later, Early Support for Java SE 6.	Java EE 1.3, 1.4 are fully supported. Early support for Java EE 5.
3.x	Java SE 5 and 6 are fully supported	Java EE 1.4, 5 are fully supported. Early support for Java EE 6.
4.x	Java SE 6, 7 are fully supported. Early support for Java SE 8.	Java EE 6 is fully supported Early support for Java EE 7
5.x	Java SE 6, 7, 8 are fully supported Early Support for Java SE 9	Java EE 6, 7 are fully supported



We consider Spring 4.3.1 Release for this training program.



- **The Dependency Inversion Principle**
  - High level modules should not depend upon low level modules. Both should depend upon abstractions.
  - Abstractions should not depend upon details, details should depend upon abstractions.
    - Depend upon Abstractions, Do not depend upon Concrete Classes.
- Based on the Hollywood principle: **Don't call me. I'll call you**

- ***As an IOC Container***
  - Creates objects and makes them available to your application
- ***As a Light-weight Framework for building Java Applications***
  - Provides an infrastructure of classes that make it easier to accomplish tasks
  - you can use Spring to build any application in Java
    - (e.g., stand-alone, Web, JEE applications, etc.), unlike many other frameworks such as Apache Struts, which is limited to web applications.
- Non Invasive & Portable
- Fully modularized (High decoupling)
- Considered as an alternative / replacement for the Enterprise JavaBean (EJB) Model
- Open source Application Framework and no vendor lock-in.
- **Most business objects in Spring apps do not depend on the Spring framework.**

# What is Spring all about ?



- A lightweight framework that addresses each tier in a Web application.
  - **Presentation layer**
    - An MVC framework that is most similar to Struts but is more powerful and easy to use.
  - **Business layer**
    - Lightweight IoC container and AOP support (including built in aspects)
  - **Persistence layer** – DAO template support for popular ORMs and JDBC
    - Simplifies persistence frameworks and JDBC
    - Complimentary: Not a replacement for a persistence framework

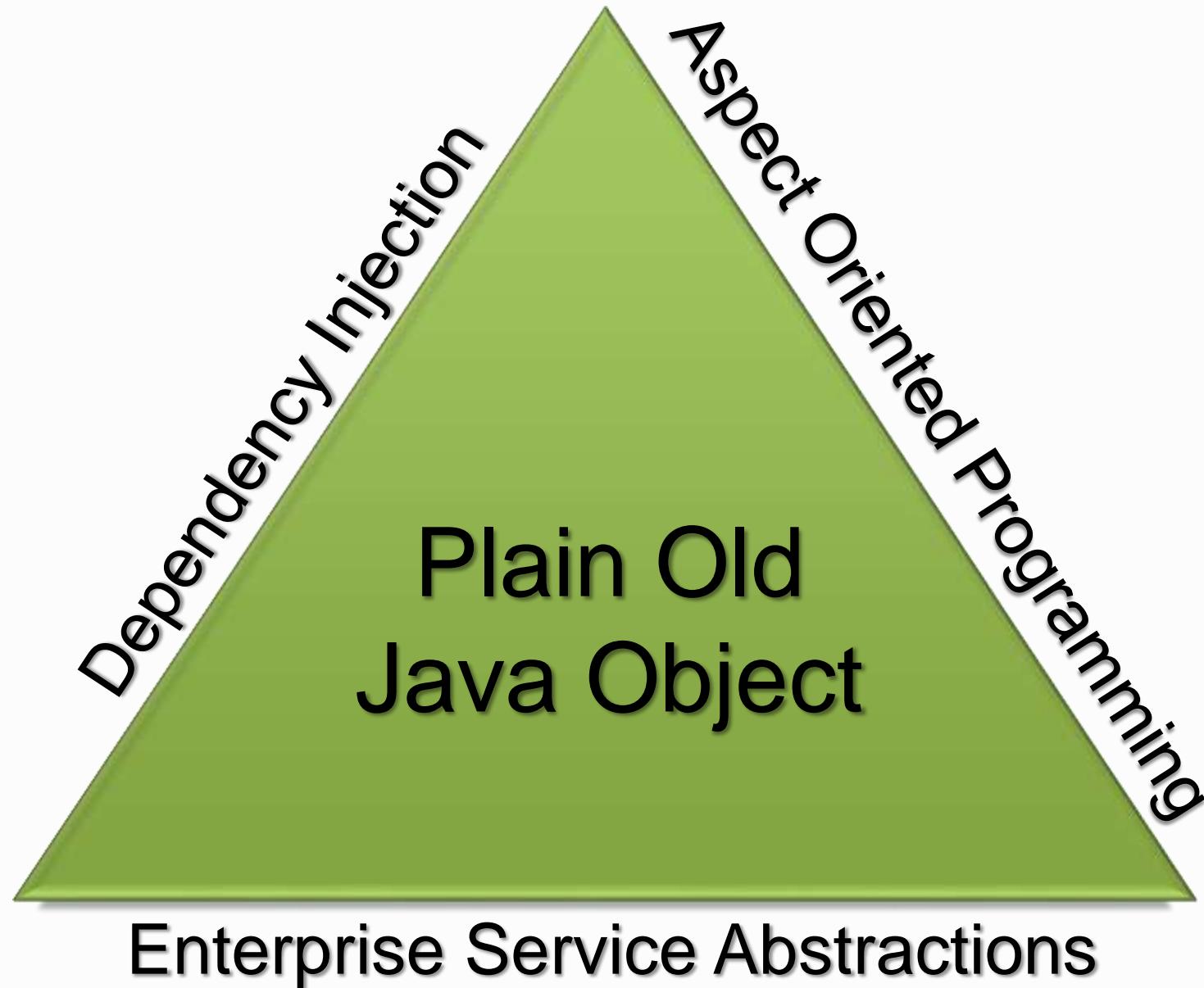
- **Well-Structured**
  - Spring code base is proven to be well structured (possibly the best)
  - 372 packages, 3500+ classes (as of 4.3.1 RELEASE)
- **Ensures Minimal Impact**
  - the lightweight part of the description doesn't really refer to the number of classes or the size of the distribution, but rather, it defines the principle of the Spring philosophy as a whole—that is, minimal impact. Spring is lightweight in the sense that you have to make few, if any, changes to your application code to gain the benefits of the Spring core, and should you choose to discontinue using Spring at any point, you will find doing so quite simple.
- **Flexible & No dependency cycles**
  - Programmers decide how to program
  - An open source framework created to address the complexity of enterprise application development.
  - Great documentation and community support

# What does Spring offer ?



- **Lightweight container and framework**
  - Most of your code will be unaware of the Spring framework
  - Use only the parts you want
- **Manages dependencies between your objects**
  - Encourages use of interfaces
  - Lessens “coupling” between objects
- **Cleaner separation of responsibilities**
  - Put logic that applies to many objects in one single place
  - Separate the class’s core responsibility from other duties
- **Simplifies database integration**
  - Spring JDBC
  - Hibernate
  - Java Persistence
- **Easily Testable**





# What Spring doesn't do



- No logging API
- Logging is a very important dependency for Spring because
  - it is the only mandatory external dependency,
  - Everyone likes to see some output from the tools they are using, and
  - Spring integrates with lots of other tools all of which have also made a choice of logging dependency. One of the goals of an application developer is often to have unified logging configured in a central place for the whole application, including all external components.
- No connection pools
- No O/R mapping layer
- It does not reinvent the wheel. There are great projects that solve these problems and Spring's goal is to make them easy to use.



*“Spring’s main aim is to make J2EE easier to use and promote good programming practice. It does this by enabling a POJO-based programming model that is applicable in a wide range of environments.”*

*– Rod Johnson*

*“Spring is an **open source**, **light-weight**, **loosely-coupled**, **aspect-oriented**, **dependency-injection** based java - jee framework software to develop all kinds of java, jee core applications by getting abstraction layer on java, jee core technologies.”*

The Spring logo, featuring a stylized green leaf icon followed by the word "spring" in a lowercase sans-serif font, with "by Pivotal" written in smaller text below it.

# Benefits of using Spring Framework



- Spring enables developers to develop enterprise-class applications using POJOs.
- Comprehensive and Modular.
- Spring does not reinvent the wheel instead, it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, other view technologies.
- Testing an application written with Spring is simple. By using JavaBean-style POJOs, it becomes easier to use dependency injection for injecting test data.
- Spring's web framework is a well-designed web MVC framework
- Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
- More and more...





*Following concepts have been covered so far...*

- Features of Bad Coding
- History of Spring
- What's Spring all about and what it provides ?
- The Spring Triangle
- What Spring doesn't do ?
- Benefits of Spring Framework



---

**GOT QUESTIONS?**

---





## TEST YOUR UNDERSTANDING



## Question #1



**TRUE**

**FALSE**

*“Spring is an open source, light-weight, tightly-coupled, aspect-oriented, dependency-injection based java - jee framework software to develop all kinds of java, jee applications by getting abstraction layer on java, jee core technologies.”*

*False*

## Question #2

Quiz



TRUE

FALSE

*Spring is a lightweight framework that addresses Presentation, Business and Persistence Layers of a Web application.*

*True*



*What is Dependency Inversion principle ?*

*High level modules should not depend upon low level modules, both should depend upon abstractions.*

*Abstractions should not depend upon details, details should depend upon abstractions.*



*What Spring doesn't do ?*

*No logging API*

*No connection pools*

*No O/R mapping layer*

## Question #5



*Why Logging is the only dependency for Spring Framework ?*

**<<TODO>>**



### *Benefits of using Spring Framework*

***Flexible & Light-weight***

***Non-Invasive***

***Portable***

***Fully Modular***

***No Dependency Cycles***

***Cleaner Separation of Responsibilities***

***Well Structured***

***Open Source***

***Great Documentation & Support***



*Give an Example for, as an Application Developer how can you get benefited from using Spring Framework:*

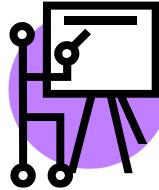
- *Make a Java method execute in a database transaction without having to deal with transaction APIs.*
- *Make a local Java method a remote procedure without having to deal with remote APIs.*
- *Make a local Java method a management operation without having to deal with JMX APIs.*
- *Make a local Java method a message handler without having to deal with JMS APIs.*



## Unit 3



# Setting up the Development Environment



## Agenda

- Setup the JDK & Eclipse
- Downloading Dependencies
- Configuring Dependencies within Eclipse IDE

# Setting up the Development Environment



- Step 1: Install Eclipse & Java Development Kit (JDK)
- Step 2: Download Spring 4.3.1 RELEASE Jars
- Step 3: Download Apache Commons Logging API
- Step 4: Download AspectJ Dependencies
- Step 5: Setup Eclipse IDE

- **Step 1: Setup Java Development Kit (JDK)**
  - You must have an Eclipse 4.x installed right under C:\SWDTOOLS\Eclipse
- **Step 2: Download and install Eclipse**
  - Download the latest version of Eclipse from <http://www.eclipse.org/downloads/>
- **Step 3: Download Spring 4.3.1 RELEASE Jars**
  - Download the Spring 4.3.1 distribution from the following site:  
<http://repo.spring.io/release/org/springframework/spring/>
- **Step 4: Download Apache Common Logging API**
  - Download the commons-logging-1.2.zip from
  - <https://mvnrepository.com/artifact/commons-logging/commons-logging/1.2>
- **Step 5: Download AspectJ Dependencies**
  - Download the aspectj folder from  
  <<Todo>>

- **Step 6: Setup Eclipse IDE & Spring Framework Libraries**
  - Extract the contents from spring-framework-4.3.1.RELEASE-dist.zip. content and navigate to spring-framework-4.3.1.RELEASE\libs folder
  - Right under Eclipse -> Preferences -> Java -> Build Path -> User Libraries -> New
  - Give a name to the User Library as: **springsource**.
    - Click on **Add External Jars** and navigate to downloaded spring-framework-4.3.1.RELEASE\libs folder and select all and Open.
    - Click on Add External Jars again and navigate to commons-logging-1.2 folder and include the Jars *commons-logging-1.2.jar* and *commons-logging-1.2-javadoc.jar* to the **springsource** user library.
  - Locate the **aspectj** folder downloaded from the previous slide.
  - Create a new User Library as: **aspectj**
    - Click on **Add External jars** and navigate to downloaded aspectj folder and select all and Open.
  - You are done with setting up the Eclipse for running Spring Applications !!!

# Setting up Eclipse IDE



The screenshot illustrates the configuration of a User Library in Eclipse IDE. On the left, the 'User Libraries' preference page is open, showing a 'New User Library' dialog where the name 'springsource' is entered. On the right, a 'JAR Selection' dialog is displayed, listing various JAR files from the 'spring-framework-4.3.1...' directory, including 'spring-aop-4.3.1.RELEASE.jar' and 'spring-beans-4.3.1.RELEASE.jar'. The 'File name:' dropdown at the bottom shows 'spring-aspects-4.3.1.RELEASE-javadoc.jar'.

Preferences

User Libraries

User library name: springsource

System library (added to the boot class path)

New... Edit... Add JARs... Add External JARs... Remove Up Down Import... Export...

user libraries

Java Build Path User Libraries JavaScript Include Path User Libraries

Preferences

User Libraries

User library name: springsource

System library (added to the boot class path)

New... Edit... Add JARs... Add External JARs... Remove Up Down Import... Export...

user libraries

Java Build Path User Libraries JavaScript Include Path User Libraries

JAR Selection

Name	Date modified	Type
spring-aop-4.3.1.RELEASE.jar	7/4/2016 8:50 AM	Executable
spring-aop-4.3.1.RELEASE-javadoc.jar	7/4/2016 9:06 AM	Executable
spring-aop-4.3.1.RELEASE-sources.jar	7/4/2016 9:06 AM	Executable
spring-aspects-4.3.1.RELEASE.jar	7/4/2016 8:54 AM	Executable
spring-aspects-4.3.1.RELEASE-javadoc.jar	7/4/2016 9:08 AM	Executable
spring-aspects-4.3.1.RELEASE-sources.jar	7/4/2016 9:08 AM	Executable
spring-beans-4.3.1.RELEASE.jar	7/4/2016 8:50 AM	Executable
spring-beans-4.3.1.RELEASE-javadoc.jar	7/4/2016 9:08 AM	Executable
spring-beans-4.3.1.RELEASE-sources.jar	7/4/2016 9:08 AM	Executable
spring-context-4.3.1.RELEASE.jar	7/4/2016 9:08 AM	Executable

File name: "spring-aspects-4.3.1.RELEASE-javadoc.jar" \*jar;\*.zip Open Cancel



---

**GOT QUESTIONS?**

---

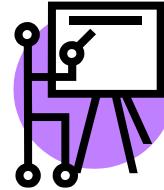




## Unit 4



# Fundamentals of Spring Framework



## Agenda

- 🎯 Architectural overview of Spring Framework
- 🎯 Factory Pattern - Revisited
- 🎯 Spring Fundamentals – Spring IOC Container
- 🎯 Checkpoint Summary
- 🎯 Quiz



## Spring Framework Runtime

### Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

### Web

WebSocket

Servlet

Web

Portlet

AOP

Aspects

Instrumentation

Messaging

### Core Container

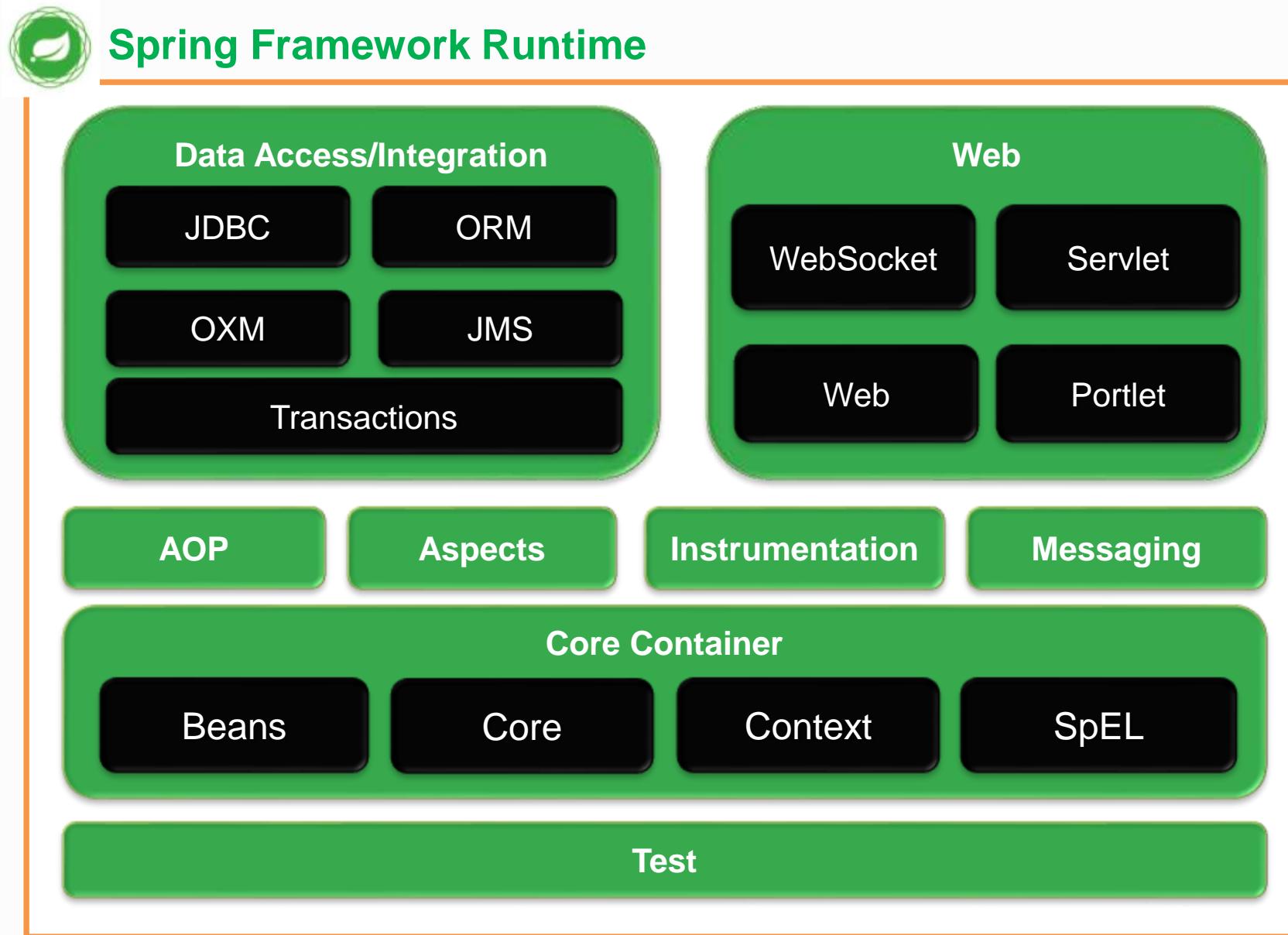
Beans

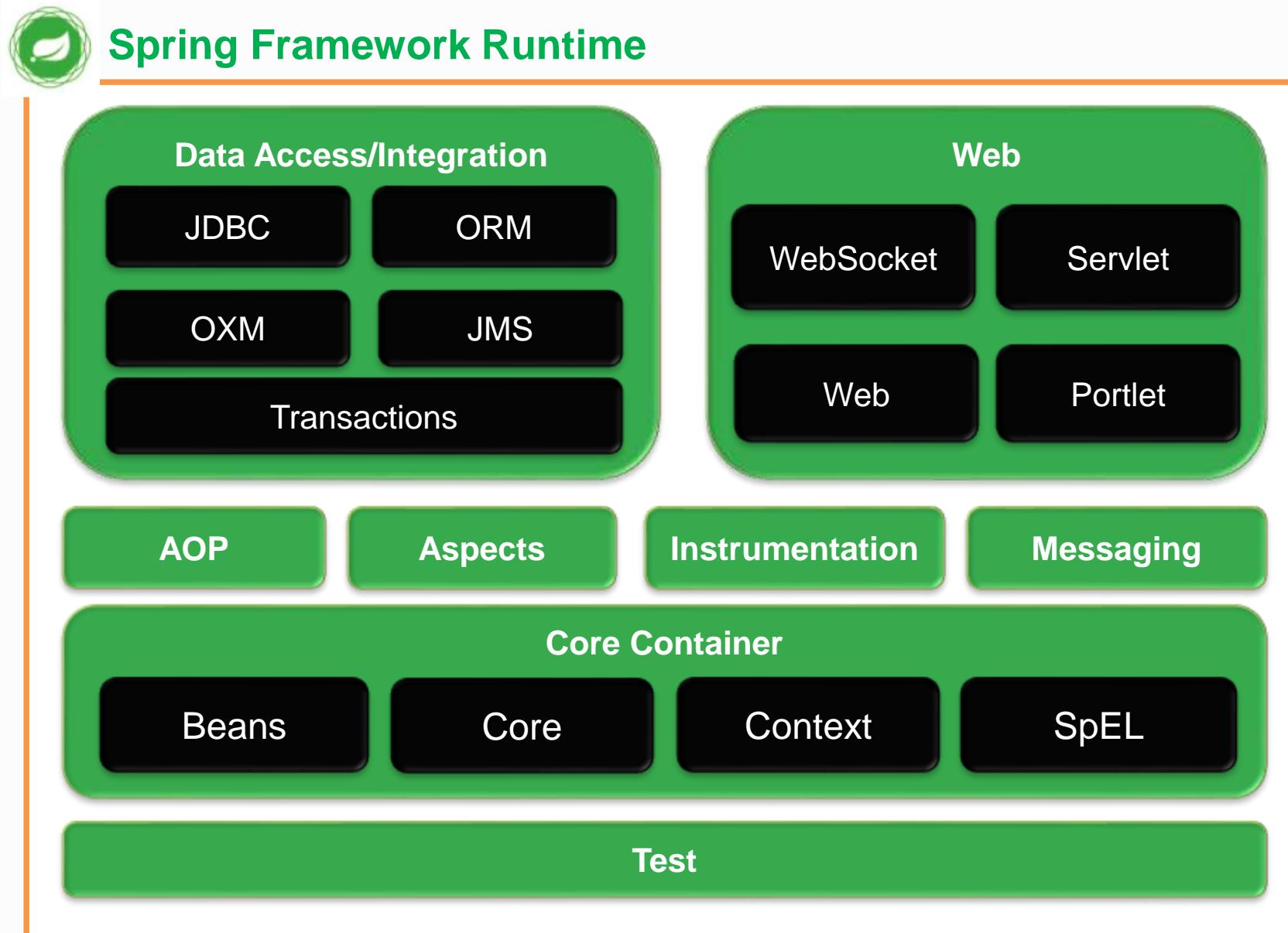
Core

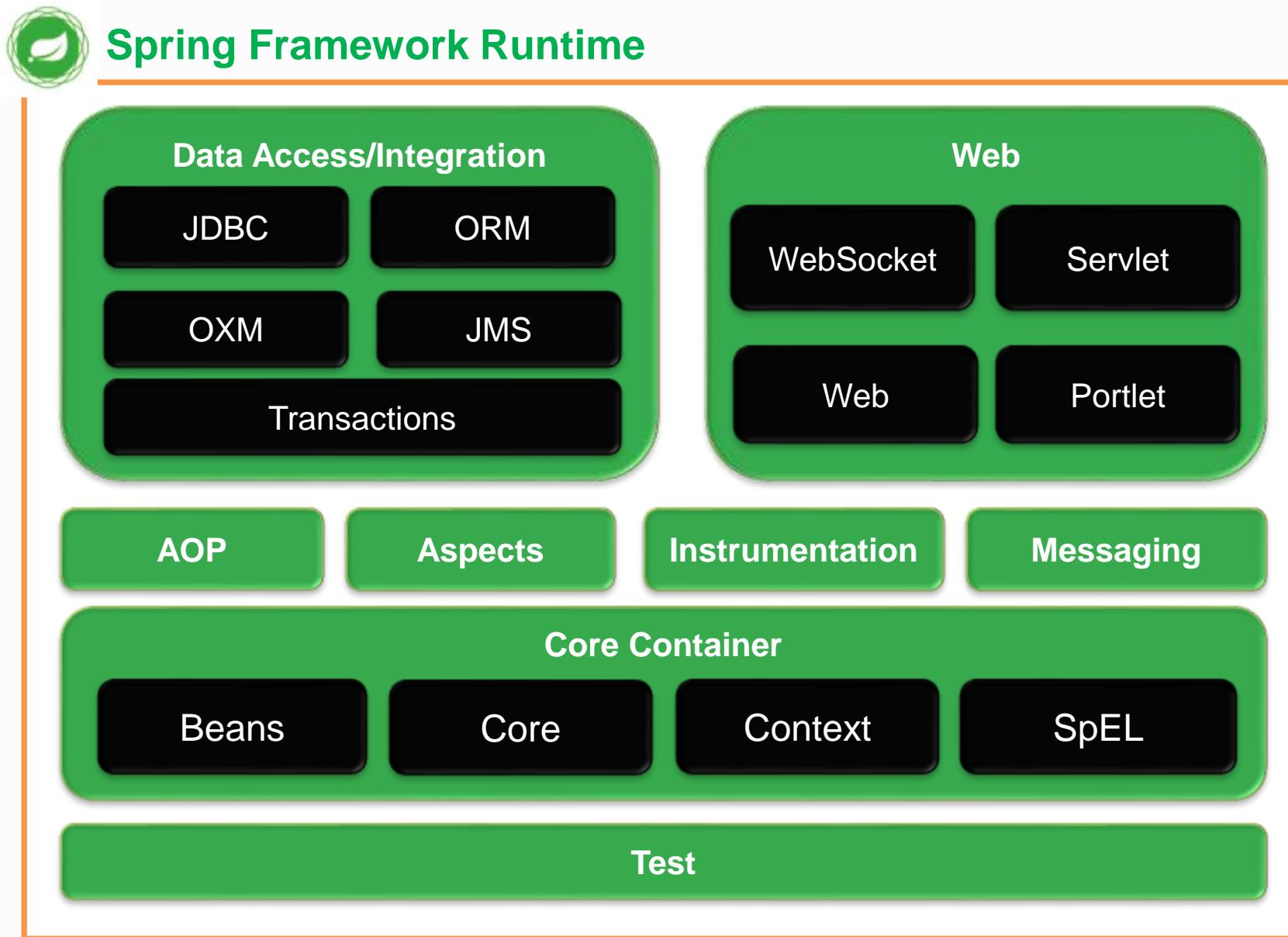
Context

SpEL

Test









## Spring Framework Runtime

### Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

### Web

WebSocket

Servlet

Web

Portlet

AOP

Aspects

Instrumentation

Messaging

### Core Container

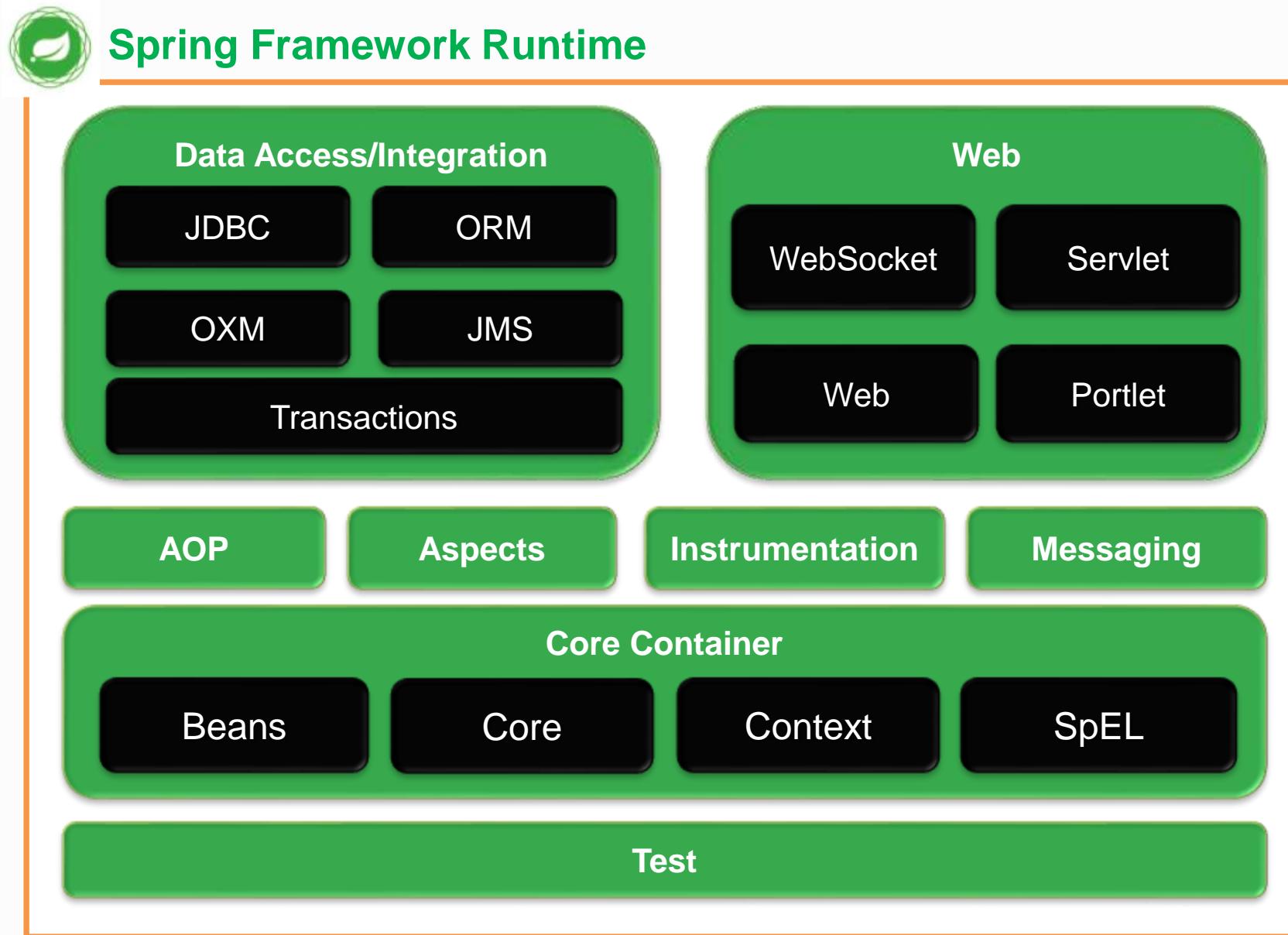
Beans

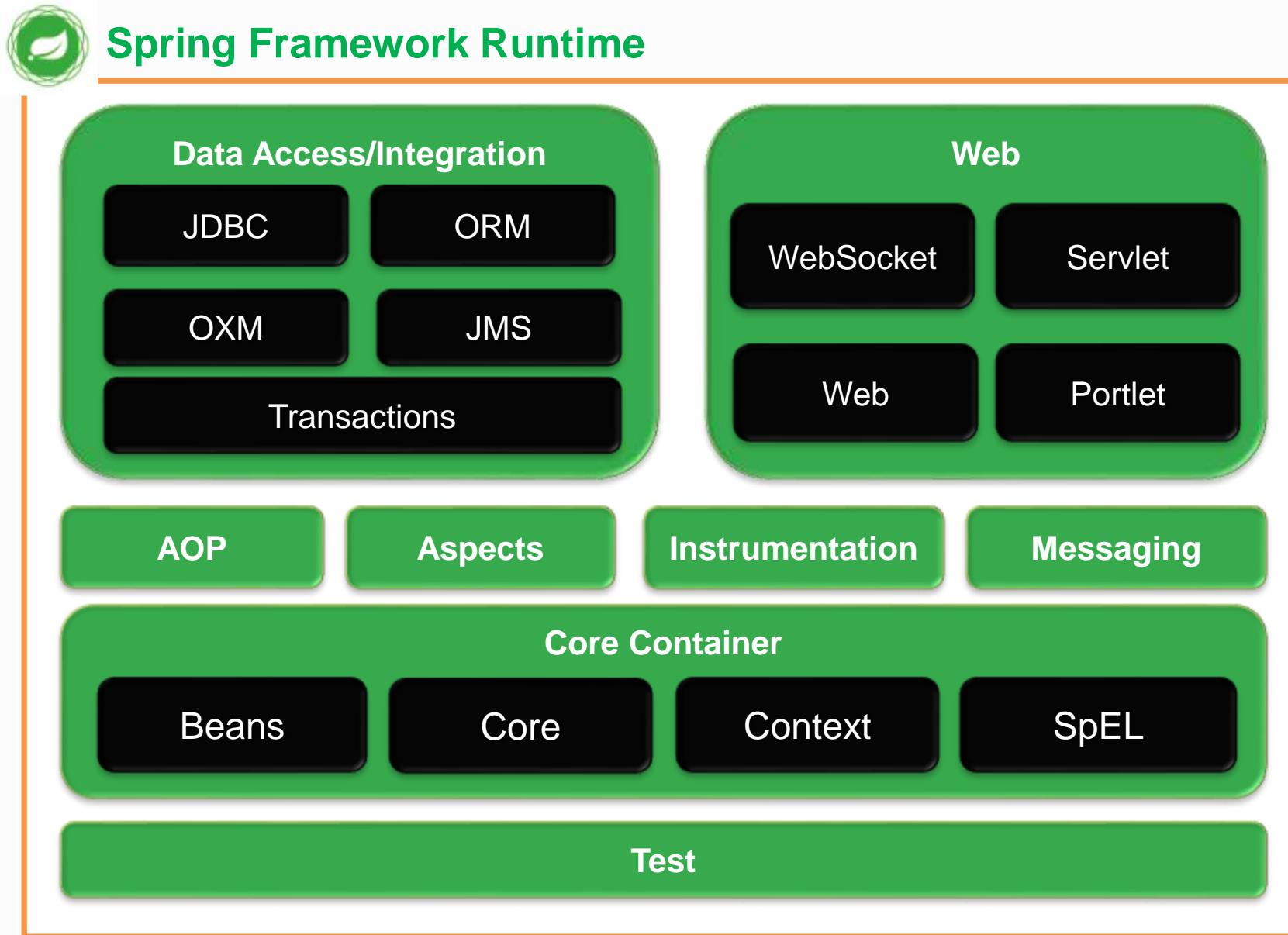
Core

Context

SpEL

Test

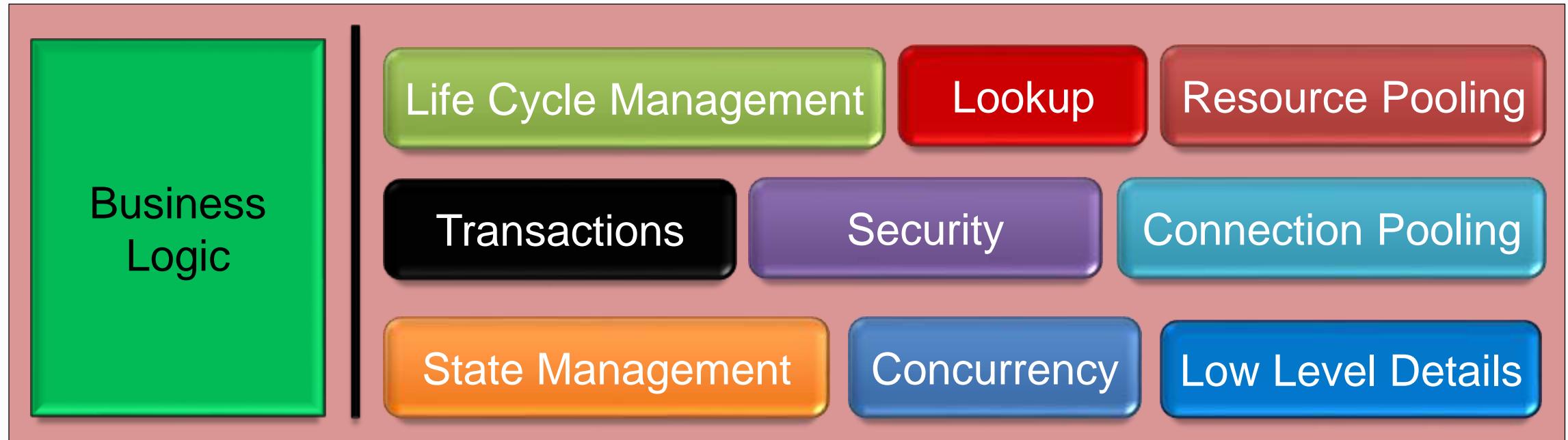




# What is a Container ?

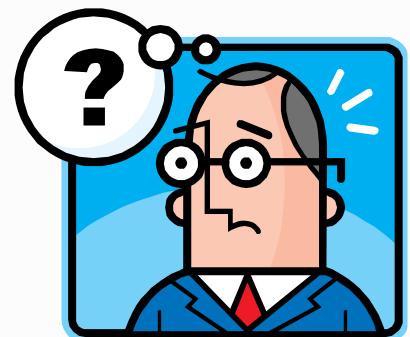


- Container, in the context of Java development, refers to a Software Application or Java Class that is responsible for managing the lifecycle of a given resource.

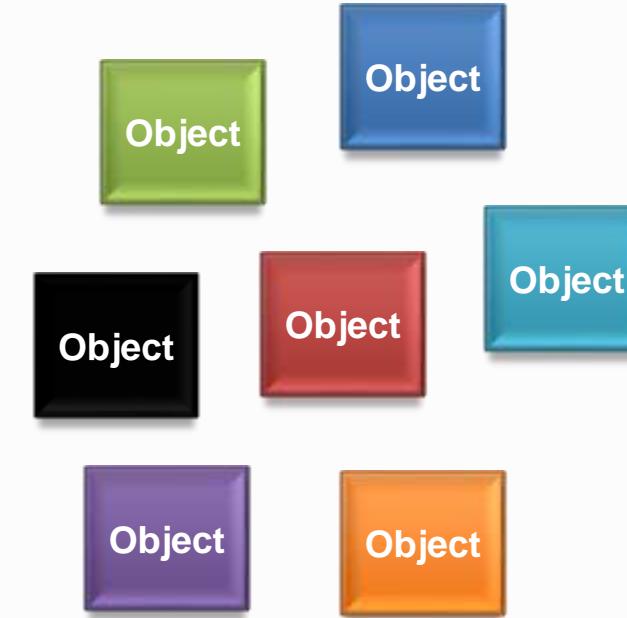
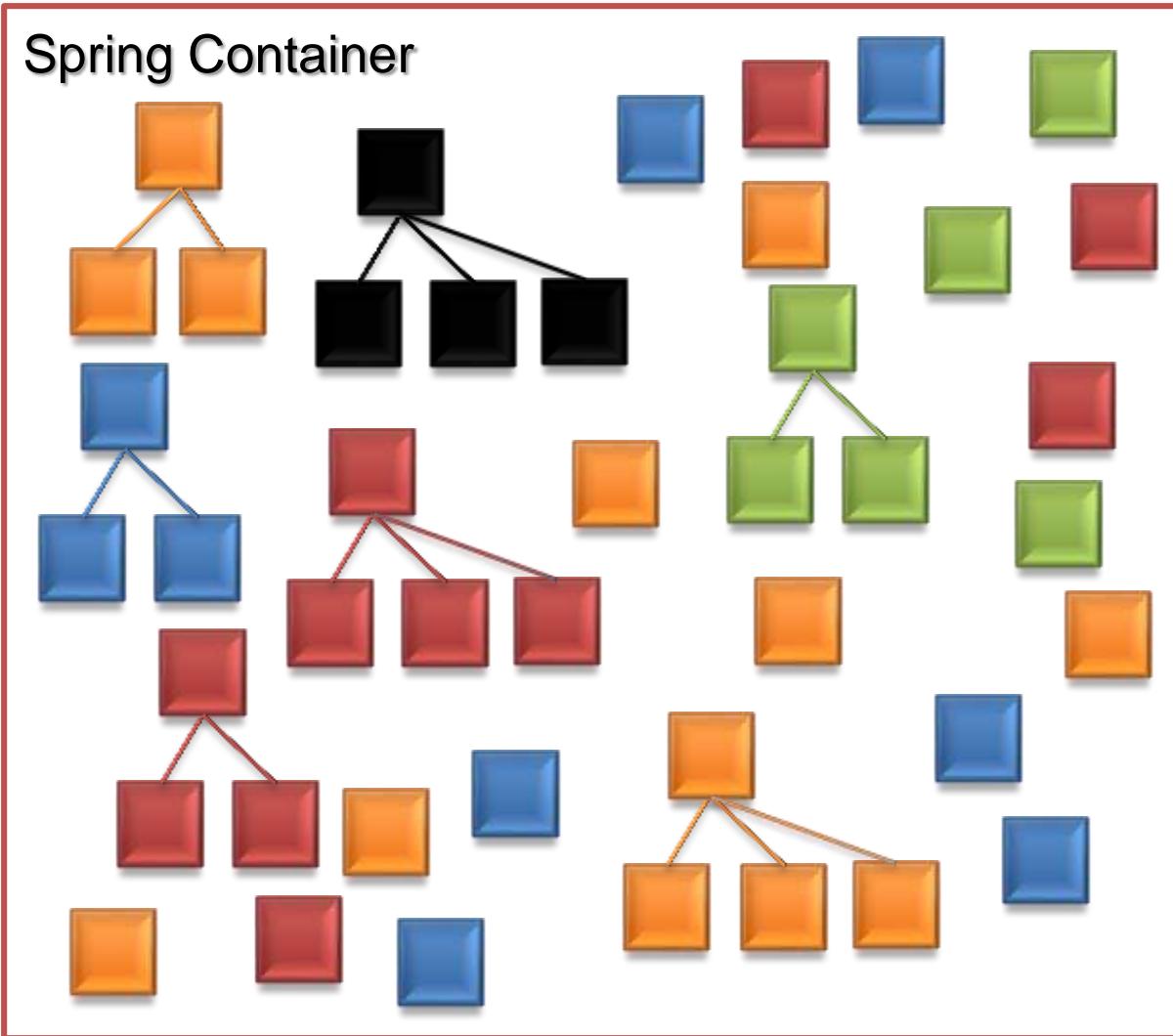


- Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand instead.
- Example Containers:-
  - Servlet Container, EJB Container, Applet Container, Spring Container

- Built-in Spring Containers
  - BeanFactory Container
  - ApplicationContext Container



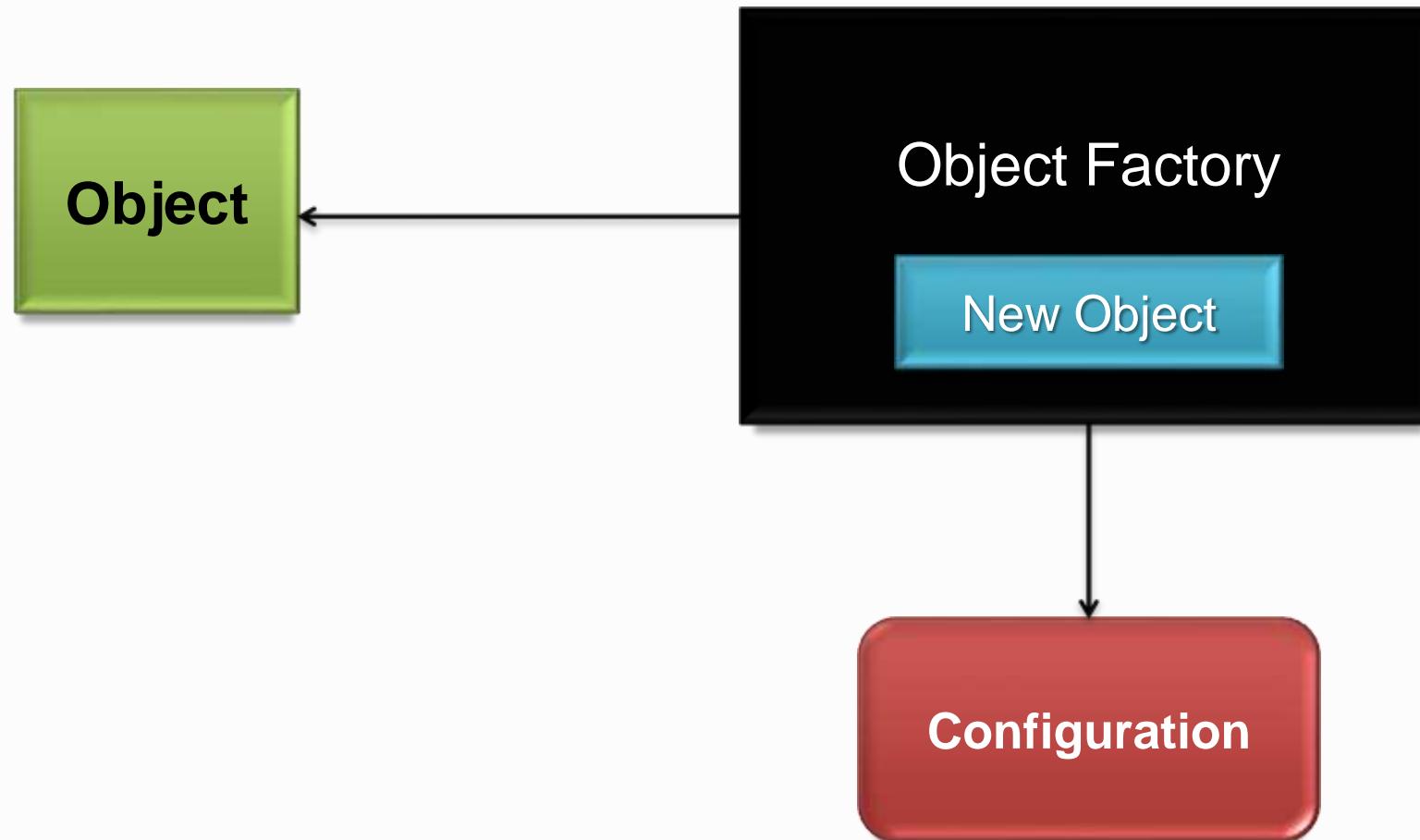
# Fundamentals of Spring – Spring Container

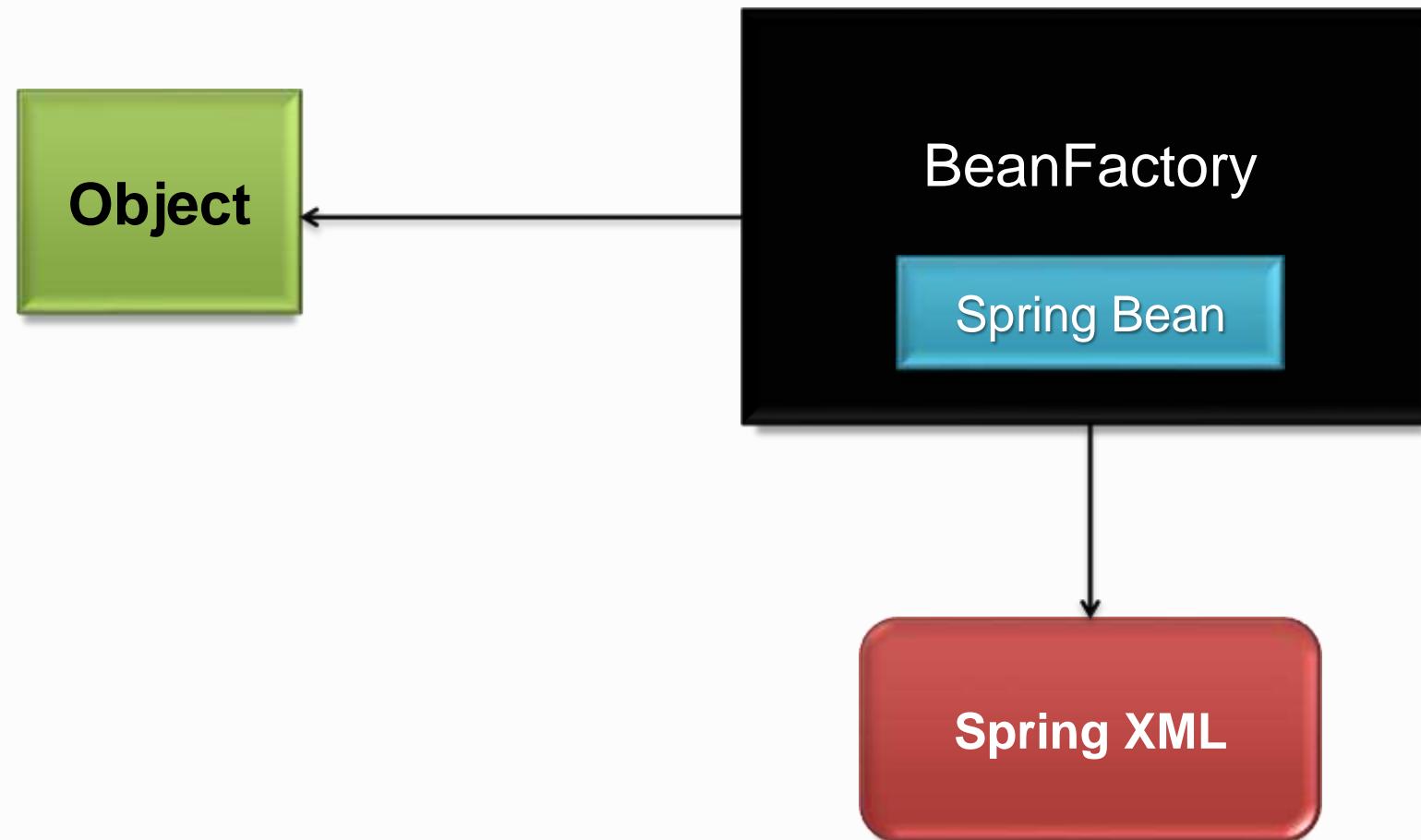


# Factory Pattern *revisited...*

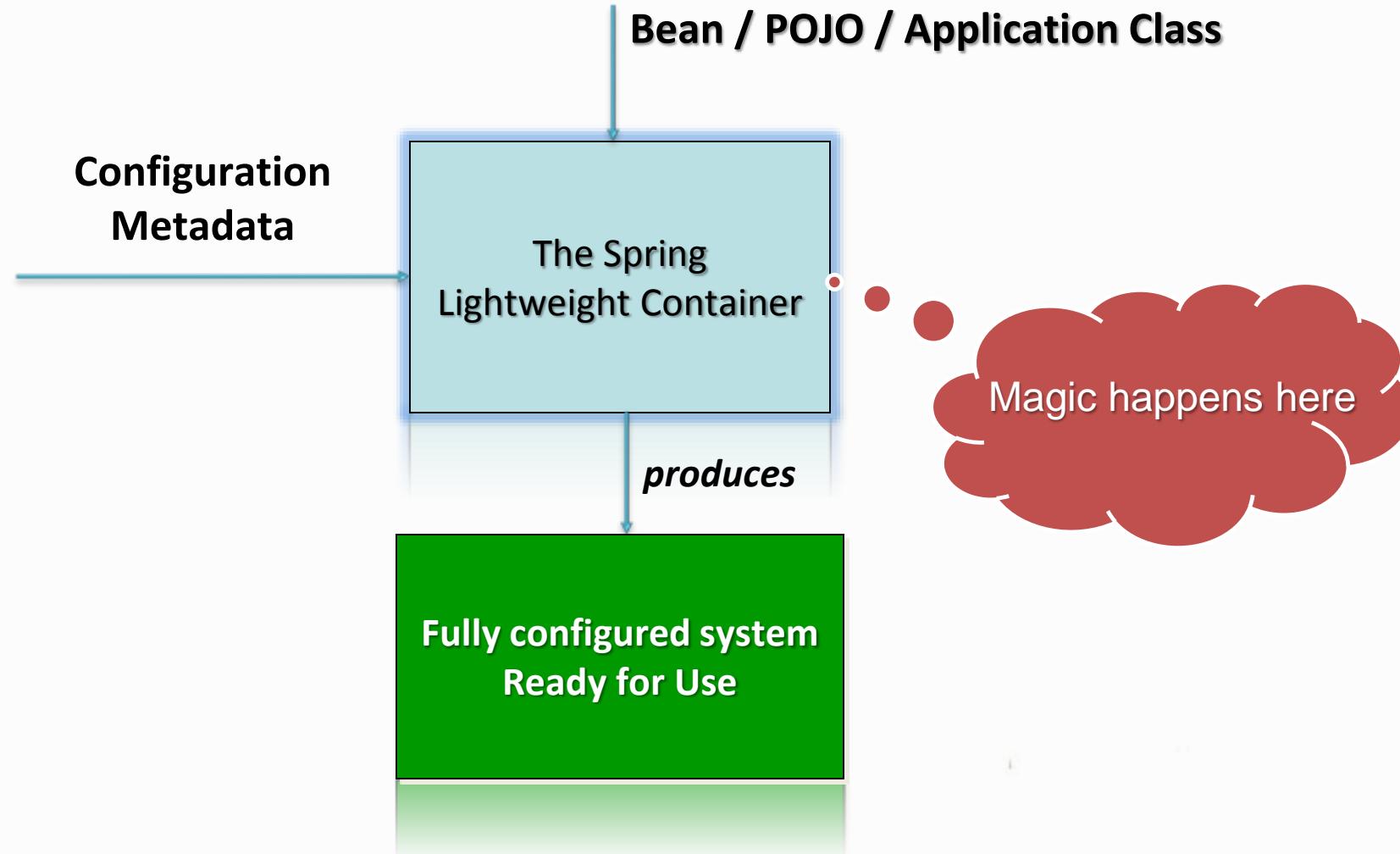


## – Factory Pattern





# Fundamentals of Spring – Spring IOC Container



- Packages that are basis for the Spring IOC Container
  - 1. Spring **BeanFactory** Container
    - **org.springframework.beans.factory.BeanFactory**
    - The BeanFactory interface provides a configuration mechanism capable of managing any type of object
  - 2. Spring **ApplicationContext** Container
    - **org.springframework.context.ApplicationContext**
    - ApplicationContext is a sub-interface of BeanFactory.
    - Provides enterprise-specific functionality.
    - It adds easier integration with:
      - Spring's AOP features
      - Message Resource handling (for use in i18n)
      - Event publication etc.

- You can instantiate a Spring IOC container using any one of the below two classes.
  - Spring Bean Factory Container
    - **org.springframework.beans.factory.BeanFactory**
  - Spring ApplicationContext Container
    - **org.springframework.context.ApplicationContext**

It's time for an exercise !!!



### *Lab 1 – Hello World Application*



Instantiating Spring Container using BeanFactory

### *Lab 2 – Hello World Application*



Accessing Property from the XML Configuration

## – Instantiating Spring Container using BeanFactory

1. Write a **HelloWorld** Java Bean with a property **message**.
2. Generate getters and setters for the property and append a sysout in the `getMessage()` to display the message: “*Hello World, Welcome to Spring !!!*”
3. Create Spring-Config.xml and add the bean definition.
4. Write Main program to instantiate the BeanFactory
5. Call the `getMessage()` method using the bean that is created in the previous step.
6. Run the program to display the output.



## – Accessing Property from the XML Configuration

1. Write a `HelloWorld` Java Bean with a property **message**.
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition.
4. Add the property **message** and set the value to: “*Hello World, Welcome to Spring !!!*”
5. Write Main program to instantiate the BeanFactory
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



- Spring's more advanced container.
- Similar to BeanFactory it can load bean definitions, wire beans together and dispense beans upon request.
- Additionally it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.
- Defined by the **org.springframework.context.ApplicationContext** interface.

- The most commonly used `ApplicationContext` implementations are:
  - `ClassPathXmlApplicationContext`
  - `FileSystemXmlApplicationContext`
  - `XmlWebApplicationContext`

## – ClassPathXmlApplicationContext

- This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.

## – FileSystemXmlApplicationContext

- This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.

## – XmlWebApplicationContext

- This container loads the XML file with definitions of all beans from within a web application.



## *Lab 3 – Hello World Application*



Using ClassPathXmlApplicationContext

## *Lab 4 – Hello World Application*



Using FileSystemXmlApplicationContext

## *Lab 5 – Hello World Application*



Using XmlWebApplicationContext (Self Study)

## – Using ClassPathXmlApplicationContext

1. Write a `HelloWorld` Java Bean with properties `name`, `message`
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition and place it under resources folder.
4. Add the properties `name`, `message` and set the values to: “`Hello`”, “`Welcome to Spring !!!`” respectively.
5. Add the resources folder to classpath.
6. Write Main program to read the Spring-Config.xml using `ClassPathXmlApplicationContext` class.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



## –Using FileSystemXmlApplicationContext

1. Write a `HelloWorld` Java Bean with properties `name`, `message`
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition.
4. Add the properties `name`, `message` and set the values to: “*Hello*”, “*Welcome to Spring !!!*” respectively.
5. Write Main program to read the Spring-Config.xml using `FileSystemXmlApplicationContext` class.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



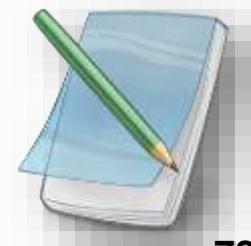
# LAB 4-1 – Hello World Application – Steps to follow



## – Using XmlWebApplicationContext

***An online example:***

<http://www.knowledgewalls.com/j2ee/books/spring-30-examples/how-to-use-xmlwebapplicationcontext-in-spring-framework-with-example>



## BeanFactory

The simplest factory, mainly for DI

### Saving Resources:

Used when resources are limited, e.g., mobile, applets etc.

### Package:

org.springframework.beans.factory.BeanFactory

### Implementation:

```
BeanFactory factory = new XmlBeanFactory  
(new ClassPathResource("wildLife.xml"));
```

## ApplicationContext

The advanced and more complex factory

Used elsewhere and has the below features,

- 1> Enterprise aware functions
- 2> Publish application events to listeners
- 3> Wire and dispose beans on request

### Package:

org.springframework.context.ApplicationContext

### Implementation:

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("wildLife.xml");
```



## ApplicationContext

Following are the additional features of Application Context:

- Provide a means for resolving text messages, including support for internationalization (i18N) of those messages, whereas BeanFactory doesn't support.
- provide a generic way to load file resources, such as images.
- publish events to beans that are registered as listeners.
- Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.
- ApplicationContext provides integration with Spring AOP.

## FINAL VERDICT:

Since ApplicationContext has many advanced features along with all the features provided by BeanFactory, it is advisable to use ApplicationContext than BeanFactory.

BeanFactory is good to use if we don't want to use any of the additional features provided by ApplicationContext.



## Bean Factory or Application Context

- *ApplicationContext* container includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*. If you are creating applications for an enterprise, choose Application Context.
- *BeanFactory* can still be used for light weight applications like mobile devices or applet based applications where data volume and speed is significant.





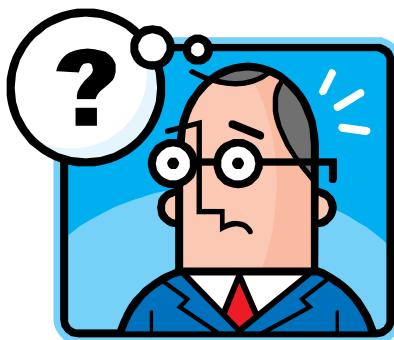
*Following concepts have been covered so far...*

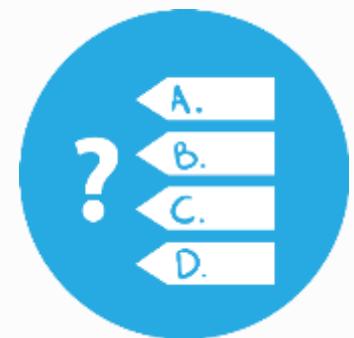
- Characteristics of Bad Code
- Architectural Overview of Spring Framework and it's components
- Factory Pattern
- Container
- Spring IOC Containers – Bean Factory & Application Context





## TEST YOUR UNDERSTANDING





*Which of the statements are correct ?*

- A - Core and beans modules provide the fundamental parts of the framework, including Dependency Injection feature.*
- B - The SpEL module provides a powerful Expression Language for querying and manipulating an object graph at runtime.*
- C - Aspects module provides integration with Aspect*

**A, B, C**



*What is Factory Pattern ?*

<<??>>



*What is Spring IOC container ?*

*The Spring IoC creates the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.*



*What are the Built-in Spring Containers ?*

***BeanFactory Container***  
***ApplicationContext Container***



### *What is BeanFactory Container ?*

*A BeanFactory is an implementation of the factory pattern that applies Inversion of Control to separate the application's configuration and dependencies from the actual application code. The most commonly used BeanFactory implementation is the XmlBeanFactory class.*

*This is the simplest container providing basic support for DI . The BeanFactory is usually preferred where the resources are limited like mobile devices or applet based applications*



*What does XMLBeanFactory do?*

***org.springframework.beans.factory.xml.XmlBeanFactory***

*Loads its beans based on the definitions contained in an XML file. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.*



*How to instantiate a BeanFactory Container ?*

```
BeanFactory factory = new XmlBeanFactory (new  
ClassPathResource("wildLife.xml"));
```



*What is ApplicationContext Container ?*

*In addition to the capabilities of a BeanFactory Container, this container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.*



*How to instantiate a ApplicationContext Container ?*

*ApplicationContext context = new  
ClassPathXmlApplicationContext("wildLife.xml");*



*What are the common implementations of the  
ApplicationContext?*

**FileSystemXmlApplicationContext**  
**ClassPathXmlApplicationContext**  
**XmlWebApplicationContext**



*BeanFactory or ApplicationContext – Which one to choose ?*

*use an ApplicationContext unless you have a really good reason for not doing so.*

## Question #12

A red rectangular button with the word "Quiz" in white. A white cursor arrow is pointing at the bottom-left corner of the button.

Quiz



*What is the rule for setting the Property for a bean in the Spring Configuration file and how can you achieve it ?*

```
<bean id="helloWorld"
      class="org.springframework.HelloWorld">
    <property name="name" value="Hello"/>
    <property name="message" value="Welcome"/>
</bean>
```



*What is Spring Configuration File ?*

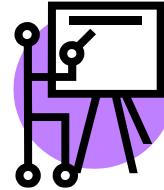
*Spring configuration file is an XML file. It contain bean definitions and other information which is used to provide context information to the Spring framework.*



## Unit 5



# Spring Bean Concepts

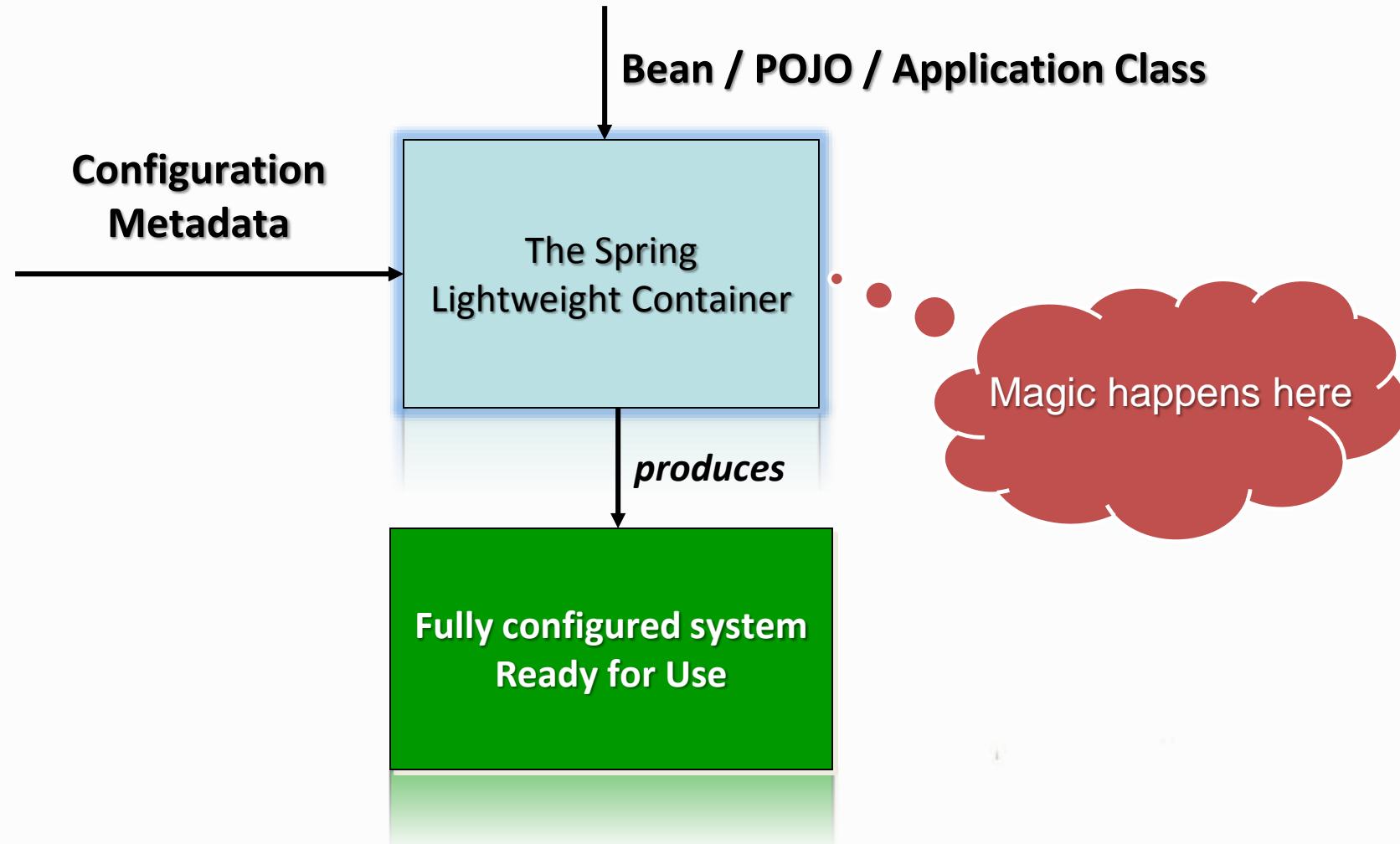


## Agenda

- Spring Bean Definition
- Naming Spring Beans
- Spring Configuration Metadata
- Instantiating Beans
- Spring Bean Aliasing
- Quiz

- The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**.
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- The bean definition contains the information called **configuration metadata** which is needed for the container to know:
  - How to create a bean
  - Bean's lifecycle details
  - Bean's dependencies
- These beans are created with the **configuration metadata** that you supply to the container, for example, in the form of XML <bean/> definitions.

# Spring IOC Container *revisited...*



# Spring Bean Definition *continued...*



Tag Name	Description	Example
<b>id</b>	Unique Id	<bean id="person" ... />
<b>name</b>	Unique Name	<bean name="lion" ... />
<b>class</b>	Fully qualified Java class name	<bean class="a.b.C" ... />
<b>scope</b>	Bean object type	<bean scope="singleton" ... />
<b>constructor-arg</b>	Constructor injection	<constructor-arg value="a" />
<b>property</b>	Setter injection	<property name="a" ... />
<b>autowire</b>	Automatic Bean referencing	<bean autowire="byName" ... />
<b>lazy-init</b>	Create a bean lazily (at its first request)	<bean lazy-init="true" ... />
<b>init-method</b>	A callback method just after bean creation	<bean init-method="log" ... />
<b>destroy-method</b>	A callback just before bean destruction	<bean destroy-method="log" ... />

- Every bean has one or more identifiers.
- Identifiers must be unique within the container that hosts the bean.
- A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered **aliases**.
- The `id` attribute allows you to specify exactly one **id**.
- If you want to introduce other aliases to the bean, you can specify them in the **name** attribute, separated by a comma (,), semicolon (;), or white space.
- You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name you must provide a name.
- Motivations for not supplying a name are related to using **inner beans** and **autowiring collaborators**.

- Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written.
- Configuration Metadata can be provided to a Spring Container in the following ways
  - through an *XML Based Configuration File*
  - through an *Annotation based Configuration*
  - through a *Java based configuration*

## – Instantiating with a Constructor

```
<bean id="exampleBean" class="examples.ExampleBean" />
<bean name="anotherExample" class="examples.ExampleBeanTwo" />
```

## – Instantiating with a Static Factory Method:

```
<bean id="clientService" class="examples.ClientService"
factory-method="createInstance"/>

public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

## – Instantiation using an instance factory method

```
<!-- the factory bean, which contains a method called createClientServiceInstance () -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>
<!-- the bean to be created via the factory bean -->
<bean id="clientService" factory-bean="serviceLocator"
factory-method="createClientServiceInstance" />
```

```
public class DefaultServiceLocator {
    private ClientService clientService = new ClientServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

## – Instantiating more than one instance factory methods

```
public class DefaultServiceLocator {  
    private static ClientService    clientService = new ClientServiceImpl();  
    private static AccountService  accountService = new AccountServiceImpl();  
    private DefaultServiceLocator() {}  
    public ClientService createClientServiceInstance() {  
        return clientService;  
    }  
    public AccountService createAccountServiceInstance() {  
        return accountService;  
    }  
}
```

---

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator" />  
<bean id="clientService" factory-bean="serviceLocator" factory-method=  
"createClientServiceInstance" />  
<bean id="accountService" factory-bean="serviceLocator" factory-method=  
"createAccountServiceInstance" />
```



## *Lab 5 – Spring Bean Creation*

- 🎯 Instantiating with a Constructor

## *Lab 6 – Spring Bean Creation*

- 🎯 Instantiating with a Static Factory Method

## *Lab 7 – Spring Bean Creation*

- 🎯 Instantiation using an instance factory method

## *Lab 8 – Spring Bean Creation*

- 🎯 Instantiating more than one Instance factory method (Self Study)

## – Steps to follow

1. Write a HelloWorld Java Bean with properties **name**, **message** and with a default constructor defined.
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition.
4. Add the properties **name**, **message** and set the values to: **"Hello"**, **"Welcome to Spring !!!"** respectively.
5. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



## – Steps to follow

1. Write a DatabaseService class with the following
  - a. `private static DatabaseService databaseService = new DatabaseService();`
  - b. a static method `getConnection()` which returns `DatabaseService` object.
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition.
4. Use the `factory-bean` and `factory-method` attributes in the Spring-Config.xml.
5. Write Main program to read the Spring-Config.xml using `ClassPathXmlApplicationContext`.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



## – Steps to follow

1. Create Transport interface that has getTransport() method.
2. Create Bus and Car classes that implement Transport interface and implement the method stated above.
3. Write a TransportService class with the following
  1. `private static TransportService transportService = new TransportService();`
  2. Create a static factory method `createService()`
  3. Create 2 instance factory methods that return Transport object:  
`createCarInstance()` and `createBusInstance()`
4. Use the **factory-bean** and **factory-method** attributes in the Spring-Config.xml.
5. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



- In a bean definition, you can supply more than one name for the bean
  - by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute.
    - `<bean id="employee" name= "emp1, emp2, emp3" ... />`
- These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.
  - `<alias name="fromName" alias="toName"/>`
- This is useful in case we need to use beans that exist in a different sub-system.

It's time for an exercise !!!



## *Lab 9 – Bean Aliasing*



Example: CoreSpring\_Lab09\_BeanAliasing



## – Steps to follow

1. Write a HelloWorld Java Bean with properties **name** , **message** .
2. Generate getters and setters for the property
3. Create Spring-Config.xml and add the bean definition.
4. Use the **name** attribute in the **<bean>** tag to define the alias of the bean.
5. Add the properties **name** , **message** and set the values to: **"Hello"**,  
**"Welcome to Spring !!!"** respectively.
6. Use the **<alias>** tag to define the alias of the bean.
7. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
8. Retrieve the values using the bean that is created in the previous step.
9. Run the program to display the output.





*Following concepts have been covered so far...*

- Spring Bean Definition
- Naming Spring Beans
- Spring Configuration Metadata
- Initializing Beans
- Bean Aliasing



QUESTION



## TEST YOUR UNDERSTANDING



## Question #1



*What is Spring Configuration Metadata ?*

<<??>>

## Question #2



*What is Spring Bean Naming Rules ?*

<<??>>



*What are the various attributes passed to a <bean> tag in a Spring Configuration File?*

- id
- name
- class
- scope
- constructor-arg
- property
- autowire
- lazy-init
- init-method
- destroy-method

## Question #4



*Difference between name and id attribute of the <bean> tag ?*

<<??>>

## Question #5



*What are all the different ways of instantiating a Spring Bean ?*

<<??>>

## Question #6



*What is Bean Aliasing ?*

<<??>>

## Question #7



*When we have name attribute of the <bean> tag, then why we have <alias> tag to define the alias of a bean ?*

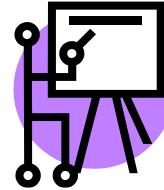
<<??>>



## Unit 6



# Dependency Injection Concepts



## Agenda

- Dependency Injection
- Types of Dependency Injection
- Various Dependencies & Configurations
- Checkpoint Summary
- Quiz

- *Dependency injection (DI)* is a process whereby objects define their dependencies.
- The container then *injects* those dependencies when it creates the bean
- This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the Service Locator pattern.
- **Advantage**
  - Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies.

- *Dependency injection (DI)* comes with two flavors
  - Constructor-based Dependency Injection
  - Setter-based Dependency Injection

- Accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

It's time for an exercise !!!



## *Lab 10 – Dependency Injection*

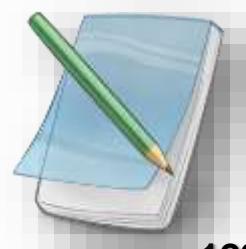


Setter Based Dependency Injection



## – Steps to follow

1. Write a Employee Java Bean with the following properties:  
**employeeName, employeeId, passportNumber, age, address.**
2. Generate getters and setters for the properties defined above.
3. Create Spring-Config.xml and add the bean definition.
4. Add the properties **employeeName, employeeId, passportNumber, age, address** in the Spring-Config.xml and set some values to them.
5. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



- Accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
- Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly.

It's time for an exercise !!!



## *Lab 11 – Dependency Injection*



Constructor-based Dependency Injection



## – Steps to follow

1. Write a Triangle Java Bean with the following properties:  
**type**, **height**.
2. Generate getters and setters for the properties defined above.
3. Create Spring-Config.xml and add the bean definition.
4. Add the properties **type**, **height** in the Spring-Config.xml and set some values to them.
5. Use the `<constructor-arg>` to re-define the properties.
6. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.
9. Use various scenarios of `<constructor-arg>` in the context



- The ‘value’ attribute is mandatory and rest are optional, e.g., ‘type’
- Constructor Injection can automatically cast the value to the desired ‘known’ type
- By default the ‘type’ of the ‘value’ is ‘java.lang.String’ (if not specified explicitly)
- Constructor injection does not interpret ordering of the arguments specified

# Constructor-based DI Ambiguity – 1



```
<bean id="myCollege" class="com.College">
<constructor-arg value="500" />
<constructor-arg value="123Abc" />
</bean>
```

```
public class College {
    private String collegeId;
    private int totalStudents;
    private String collegeAdd;

    public College(int totalStudents, String collegeId)
    {
        this.totalStudents = totalStudents;
        this.collegeId = collegeId;
    }

    public College(String collegeAdd, String collegeId)
    {
        this.collegeAdd = collegeAdd;
        this.collegeId = collegeId;
    }
}
```

Which constructor will be called?



# Constructor-based DI Ambiguity – 1 Solution



```
<bean id="myCollege" class="com.College">
<constructor-arg value="500" type="int" />
<constructor-arg value="123Abc" type="java.lang.String"/>
</bean>
```

```
public class College {
    private String collegeId;
    private int totalStudents;
    private String collegeAdd;

    public College(int totalStudents, String collegeId)
    {
        this.totalStudents = totalStudents;
        this.collegeId = collegeId;
    }

    public College(String collegeAdd, String collegeId)
    {
        this.collegeAdd = collegeAdd;
        this.collegeId = collegeId;
    }
}
```

The 'type' of the value is specified



# Constructor-based DI Ambiguity – 2



```
<bean id="myCollege" class="com.College">
<constructor-arg value="500" type="int" />
<constructor-arg value="123Abc" type="java.lang.String"/>
</bean>
```

---

```
public class College {
    private String collegeId;
    private int totalStudents;
    private String collegeAdd;

    public College(int totalStudents, String collegeId) {
        this.totalStudents = totalStudents;
        this.collegeId = collegeId;
    }

    public College(String collegeAdd, int totalStudents) {
        this.totalStudents = totalStudents;
        this.collegeAdd = collegeAdd;
    }
}
```

Which constructor will be called?



# Constructor-based DI Ambiguity – 2 Solution



```
<bean id="myCollege" class="com.College">
<constructor-arg value="500" type="int" index="0"/>
<constructor-arg value="123Abc" type="java.lang.String" index="1"/>
</bean>
```

```
public class College {
    private String collegeId;
    private int totalStudents;
    private String collegeAdd;

    public College(int totalStudents, String collegeId) {
        this.totalStudents = totalStudents;
        this.collegeId = collegeId;
    }

    public College(String collegeAdd, int totalStudents) {
        this.totalStudents = totalStudents;
        this.collegeAdd = collegeAdd;
    }
}
```

The 'index' of the value is specified



## Step 1:

The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code or annotations.

## Step 2:

For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, when the bean is actually created.

## Step 3:

Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.

## Step 4:

Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as **int**, **long**, **String**, **boolean** etc.

## – Which one to choose ?

- Setter injection gets preference over constructor injection when both are specified
- Constructor injection cannot partially initialize values
- Circular dependency can be achieved by setter injection
- Security is lesser in setter injection as it can be overridden
- Constructor injection fully ensures dependency injection but setter injection does not
- Setter injection is more readable



- Straight values (primitives, Strings, and so on)
- PropertyPlaceholderConfigurer
- Idref
- Reference to Other Beans
- Inner beans
- Collections
- Null and empty string values
- Compound property names
- depends-on
- Lazy Initialized Beans
- Multiple Configuration Files
- Resolving Text Messages



- Straight values (primitives, Strings etc)

```
<bean id="myDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroymethod="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
    <property name="username" value="root" />
    <property name="password" value="password@123" />
</bean>
```

## – PropertyPlaceholderConfigurer

```
class HelloWorld {  
    private String prefixProp;  
    private String suffixProp;  
    public String getPrefixProp() {  
        return prefixProp;  
    }  
    public void setPrefixProp(String prefixProp) {  
        this.prefixProp = prefixProp;  
    }  
    public String getSuffixProp() {  
        return suffixProp;  
    }  
    public void setSuffixProp(String suffixProp) {  
        this.suffixProp = suffixProp;  
    }  
    public void sayHello() {  
        System.out.println(prefixProp + "!");  
    }  
    public void sayGoodbye() {  
        System.out.println(suffixProp + "!");  
    }  
}
```

Load the `helloWorldBean` using `ApplicationContext` in the main class to see the output.

```
<bean  
    class="org.springframework.beans.factory.  
config.PropertyPlaceholderConfigurer">  
    <property name="location">  
        <value>constants.properties</value>  
    </property>  
    </bean>  
    <bean id="helloWorldBean"  
        class="com.services.HelloWorld">  
        <property name="prefixProp" value="${prefix}" />  
        <property name="suffixProp" value="${suffix}" />  
    </bean>
```

constants.properties

1	prefix=Hello
2	suffix=Goodbye

## – PropertyPlaceholderConfigurer *contd...*

```
<bean id="mappings" class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<!-- typed as a java.util.Properties -->
<property name="properties">
    <value>
        jdbc.driver.className=com.mysql.jdbc.Driver
        jdbc.url=jdbc:mysql://localhost:3306/mydb
    </value>
</property>
</bean>
```

## – Idref

- The idref element is simply an error-proof way to pass the id (string value - not a reference) of another bean in the container to a <constructor-arg/> or <property/> element.

```
<bean id="theTargetBean" class="..." />
<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```



```
<bean id="theTargetBean" class="..." />
<bean id="theClientBean" class="...">
    <property name="targetName"
        value="theTargetBean" />
</bean>
```



## *Lab 12 – Dependencies*

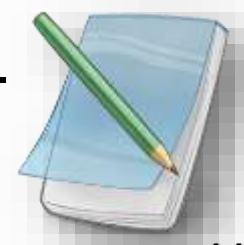


Write a program that covers the following concepts

- Straight values (primitives, Strings etc.)
- PropertyPlaceholderConfigurer
- idref

## – Steps to follow

1. Write a DatabaseConnection Java Bean with the following properties:  
**connection, url.**
2. Generate getters and setters for the properties defined above.
3. Create connection.properties and add the following to it.  
**jdbc.driver.className=com.mysql.jdbc.Driver**  
**jdbc.url=jdbc:mysql://localhost:3306/mydb**
4. Configure PropertyPlaceholderConfigurer bean in the Spring-Config.xml.
5. Create Spring-Config.xml and add the bean definition and add the properties  
**connection, url** in the Spring-Config.xml and set the values.
6. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



- References to Other Beans
  - Used to set properties that reference other beans
  - The ref element is the sub element inside a <constructor-arg> or <property> definition element.
  - The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may be initialized already by the container.)
  - All references are ultimately a reference to another object.
  - Scoping and validation depend on whether you specify the id/ name of the other object through the **bean**, **local**, or **parent** attributes.

## – References to Other Beans

### – `<ref bean="someBean" />`

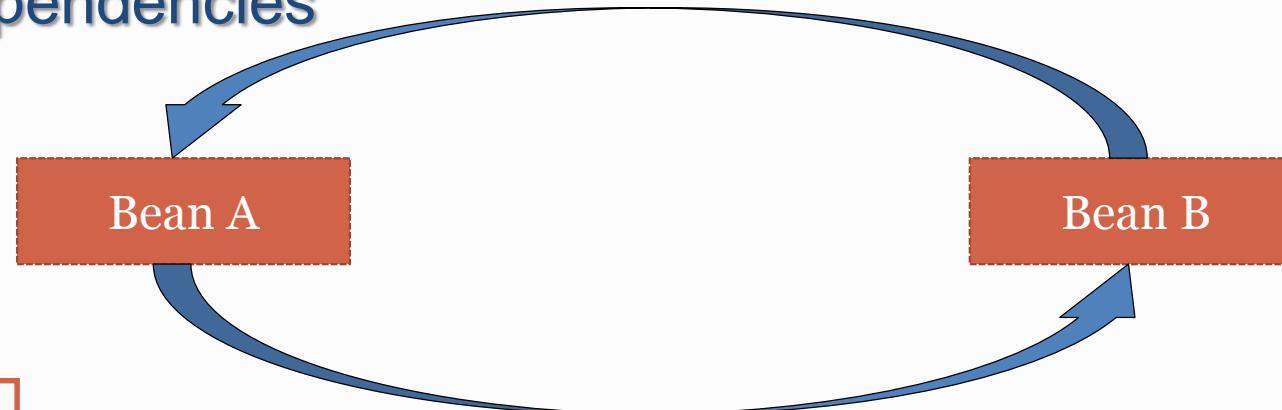
– Specifying the target bean through the `bean` attribute of the `<ref>` tag is the most general form, and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. *The value of the bean attribute may be the same as the id attribute of the target bean, or as one of the values in the name attribute of the target bean.*

### – `<ref local="someBean" />`

– Specifying the target bean through the `local` attribute leverages the ability of the XML parser to validate XML id references within the same file. *The value of the local attribute must be the same as the id attribute of the target bean.* The XML parser issues an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors as early as possible) if the target bean is in the same XML file.

### – `<ref parent="someBean" />`

- References to Other Beans
  - Circular Dependencies



```
public class A {  
    private B b;  
    public A (B b) {  
        this.b = b;  
    }  
}
```

**BeanCurrentlyInCreationException**

```
<bean id="a" class="A">  
    <constructor-arg ref="b" />  
</bean>  
  
<bean id="b" class="B">  
    <constructor-arg ref="a" />  
</bean>
```

```
public class B {  
    private A a;  
    public B (A a) {  
        this.a = a;  
    }  
}
```

## – Inner Beans

- Inner bean is also a bean reference.
- A bean defined within another bean.
  - A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements defines a so-called inner bean.
- The inner bean is fully local to the outer bean.
- The inner bean has *prototype* scope.
- An inner bean definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag.
- Inner beans are always anonymous and they are always created with the outer bean.
- It is not possible to inject inner beans into collaborating beans other than into the enclosing bean.

# Dependencies and Configurations – Inner Beans contd...



```
<bean id="triangle"
      class="org.springbasics.Triangle">
    <property name="pointA" ref="point1" />
    <property name="pointB" ref="point2" />
    <property name="pointC" ref="point3" />
</bean>
<bean id="point1" class="org.springbasics.Point">
    <property name="x" value="98" />
    <property name="y" value="99" />
</bean>
<bean id="point2" class="org.springbasics.Point">
    <property name="x" value="97" />
    <property name="y" value="98" />
</bean>
<bean id="point3" class="org.springbasics.Point">
    <property name="x" value="96" />
    <property name="y" value="97" />
</bean>
```

```
<bean id="triangle"
      class="org.springbasics.Triangle">
    <property name="pointA" ref="point1" />
    <property name="pointB">
        <!-- Inner Beans with bean id tag skipped.-->
        <!-- id="point2" -->
        <bean class="org.springbasics.Point">
            <property name="x" value="97" />
            <property name="y" value="98" />
        </bean>
    </property>
    <property name="pointC">
        <!-- Inner Beans with bean id tag skipped.-->
        <!-- id="point3" -->
        <bean class="org.springbasics.Point">
            <property name="x" value="96" />
            <property name="y" value="97" />
        </bean>
    </property>
</bean>
<bean id="point1" class="org.springbasics.Point">
    <property name="x" value="98" />
    <property name="y" value="99" />
</bean>
```



## *Lab 13 – Dependencies*



- Write a program that covers the following concepts
- References to other beans
  - Inner Beans

## – Steps to follow

1. Take the example LAB 12 – Dependencies – PropertyPlaceholderConfigurer and copy the contents in to a new Java Project.
2. Write a Employee Java Bean with the following properties:  
**employeeName, employeeId, passportNumber, age.**
3. Write a Address Java Bean with the following properties:  
**doorNumber, street, city, state, country, pincode.**
4. Generate getters and setters for the properties defined above.
5. Create Spring-Config.xml and add the bean definitions and add the properties and set the values.
6. Add DatabaseConnection Bean as a reference to the Employee Bean.
7. Add Address as an Inner Bean to the Employee Bean.
8. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext.**
9. Retrieve the values using the bean that is created in the previous step.
10. Run the program to display the output.



## – Collections

- In the `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements, you set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively.

Tag Name	Inner Tag Name	Java Collection Type	Specification
<code>&lt;list&gt;</code>	<code>&lt;value&gt;</code>	<code>java.util.List&lt;E&gt;</code>	Allows duplicate entries
<code>&lt;map&gt;</code>	<code>&lt;entry&gt;</code>	<code>java.util.Map&lt;K, V&gt;</code>	Key-Value pair of any object type
<code>&lt;set&gt;</code>	<code>&lt;value&gt;</code>	<code>java.util.Set&lt;E&gt;</code>	Does not allow duplicate entries
<code>&lt;props&gt;</code>	<code>&lt;prop&gt;</code>	<code>java.util.Properties</code>	Key-Value pair of type ‘String’

- An <entry> in a <map> is made up of a key and a value, either of which can be a primitive value or a reference to another bean.

Element	Description
<b>key</b>	Specifies the key of the map entry as a String
<b>Key-ref</b>	Specifies the key of the map entry as a reference to a bean in the Spring context
<b>value</b>	Specifies the value of the map entry as a String
<b>value-ref</b>	Specifies the value of the map entry as a reference to a bean in the Spring context

## – <props> and <list> usage

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <property name="blackList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
    ...
    ...

```

## – <map> and <set> usage

```
...
...
<property name="mediaTypes">
    <map>
        <entry key="atom" value="application/atom+xml" />
        <entry key="html" value="text/html" />
        <entry key="json" value="application/json" />
    </map>
</property>

<property name="places">
    <set>
        <value>Kolkata</value>
        <value>Hyderabad</value>
        <value>Bengaluru</value>
    </set>
</property>
```

# Consider an example – Class Member Variable



## Drawing Class

```
Shape  
draw()
```

## Different Class

```
Circle  
draw()
```

## Drawing Class

```
public class Drawing {  
    private Shape shape;  
    public void drawShape() {  
        this.shape.draw();  
    }  
    public void setShape(Shape shape) {  
        this.shape = shape;  
    }  
}
```

## Different Class

```
Circle circle = new Circle();  
drawing.setShape(circle);  
drawing.drawShape();
```

## – Null and Empty values

- Spring treats empty arguments for properties and the like as empty Strings.
- The following XML-based configuration metadata snippet sets the email property to the **empty** String value ("")

```
<bean class="ExampleBean">  
  <property name="email" value="" />  
</bean>
```



```
exampleBean.setEmail("");
```

- The following XML-based configuration metadata snippet sets the email property to **null** value

```
<bean class="ExampleBean">  
  <property name="email">  
    <null />  
  </property>  
</bean>
```



```
exampleBean.setEmail(null);
```

- Compound property names

```
<bean id="country" class="example.India">
    <property name="person.age" value="23" />
</bean>
```

- “country” bean must have a “person” nested bean, which in turn has an “age” property
- Equivalent to countryBean.getPerson().setAge(23)



## *Lab 14 – Dependencies*



- Write a program that covers the following concepts
- Collections
  - Null & Empty values

## *Lab 14-1 – Dependencies*



- Write a program that covers the following concepts
- Compound Property Values

## – Steps to follow

1. Write a Employee Java Bean with the following properties:  
**employeeName, employeeId, passportNumber, age.**
2. Write a CollectionImplementer Java Bean with the following properties:  
**lists, sets, maps, properties** of type List, Set, Map and Properties respectively.
3. Generate getters and setters for the properties defined above.
4. Create Spring-Config.xml and add the Employee bean definition and add the properties and set the values.
5. Create collectionImplementerBean in the Spring-Config.xml and add collections using **<list>, <set>, <map>** and **<prop>** tags.
6. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



## – Steps to follow

1. Write a Employee Java Bean with the following properties:  
**employeeName, employeeId, passportNumber, age.**
2. Write a Address Java Bean with the following properties:  
**doorNumber, street, city, state, country, pincode.**
3. Generate getters and setters for the properties defined above.
4. Create Spring-Config.xml and add the Employee bean definitions and add the properties and set the values. Use Null and Empty values for some of the properties.
5. Define the Address Bean to the Spring-Config.xml and add it as a reference in the Employee Bean. Set the values for the Address Bean using Compound properties with in the Employee Bean using **<property>** tag.
6. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



## – Using depends-on

```
<bean id="beanOne" class="ExampleBean" depends-on="manager" />
<bean id="manager" class="ManagerBean" />
```

- To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute, with commas, whitespace and semicolons, used as valid delimiters:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
<property name="manager" ref="manager" />
</bean>
<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

## – Lazy-initialized beans

- By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.
- you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized.
- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
- In XML, this behavior is controlled by the `lazy-init` attribute on the `<bean>` element; for example:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true" />
<bean name="not.lazy" class="com.foo.AnotherBean" />
```

## – Lazy-initialized beans *contd...*

- You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the `<beans/>` element;

```
<?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.xsd"
       default-lazy-init="true">
    ...
    ...
    ...
    <!-- no beans will be pre-instantiated... -->
    ...
    ...
    ...
</beans>
```



## *Lab 15 – Dependencies*



- Write a program that covers the following concepts
- using depends-on

## – Steps to follow

1. Copy the Lab 12 – PropertyPlaceholderConfigurer and create a new Java Project.
2. Write a **Address** Java Bean with the following properties:  
**doorNumber, street, city, state, country, pincode.**
3. Write a **Employee** Java Bean with the following properties: **employeeName, employeeId, passportNumber, age, Address, DatabaseConnection**
4. Generate getters and setters for the properties defined above.
5. Create Spring-Config.xml and add the bean definitions, add the properties and set the values and add the **depends-on** tag to the employee bean and set the value as **databaseConnection**.
6. Define the **Address, DatabaseConnection** beans as references in the **Employee** Bean.
7. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
8. Retrieve the values using the bean that is created in the previous step.
9. Run the program to display the output.

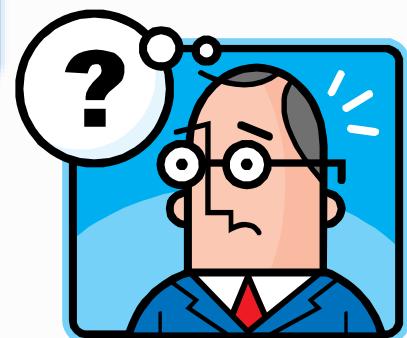


## – Multiple Configuration Files

- In a large project structure, the Spring's bean configuration files are located in different folders for easy maintainability and modular.
- For example, Spring-Common.xml in *common* folder, Spring-Connection.xml in *connections* folder, Spring-ModuleA.xml in *ModuleA* folder and so on...

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext(new String[]  
    {"Spring-Common.xml", "Spring-Connection.xml",  
    "Spring-ModuleA.xml"});
```

***Is this a right approach ?***



- And put all these XMLs in the class path.
  - project-classpath/Spring-Common.xml
  - project-classpath/Spring-Connection.xml
  - project-classpath/Spring-ModuleA.xml

- Multiple Configuration Files *contd...*
  - Create a Spring-All-Module.xml file, and import the entire Spring bean files like this

## Spring-All-Module.xml

```
<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.xsd">

    <import resource="common/Spring-Common.xml"/>
    <import resource="connections/Spring-Connection.xml"/>
    <import resource="moduleA/Spring-ModuleA.xml"/>

</beans>
```



- Now you can load a single XML like this:

```
ApplicationContext context = new ClassPathXmlApplicationContext(new String[]
        {"Spring-All-Module.xml"});
```

- Resolving Text Messages
  - Internationalization using `MessageSource`
    - The `ApplicationContext` interface extends an interface called `MessageSource`, and therefore provides internationalization (i18n) functionality.
    - The methods defined in this interface include:
      - `String getMessage(String code, Object[] args, String default, Locale loc)`
      - `String getMessage(String code, Object[] args, Locale loc)`:
  - Consider an example



## *Lab 16 – Dependencies*



- Write a program that covers the following concepts
- Multiple Configuration Files
  - Resolving Text Messages

## – Steps to follow

1. Create the files:

messageBundle\_en\_US.properties

Hello.User=Welcome {0}, Your Last logon attempt was on {1}

Welcome.Message=Good Morning !!!

messageBundle\_es\_ES.properties

Hello.User=Bienvenida {0}, Su intento de inicio de sesión  
Última estaba enc {1}

Welcome.Message=Hola, buenos días !!!

messageBundle\_fr\_FR.properties

Hello.User=Accueil {0}, Votre tentative de connexion  
dernière était sur {1}

Welcome.Message=Bonjour, bonjour !!!

2. Create Spring-Config.xml and configure **ResourceBundleMessageSource**
3. Read the Spring-Config.xml in the Main class and use getMessage() to retrieve the values from the property files created and configured in the previous steps.





*Following concepts have been covered so far...*

- Dependency Injection
- Different Types of Dependency Injection
- Various Dependencies Configurations



---

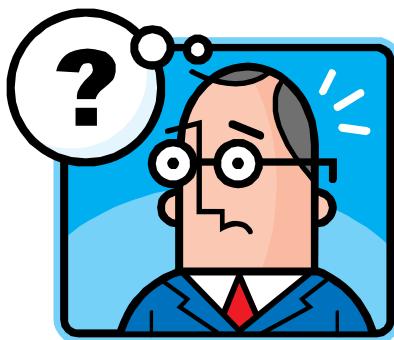
**GOT QUESTIONS?**

---





## TEST YOUR UNDERSTANDING





*What is Dependency Injection ?*

*TODO*

## Question #2



**TRUE**

**FALSE**

*Setter Injection is preferred over Constructor Injection.*

*False*



**TRUE**

**FALSE**

**depends-on** attribute of the <bean> tag is to express a dependency on multiple beans by supplying a list of bean names as the value of the depends-on attribute, with commas, whitespace and semicolons, used as valid delimiters

*True*

## Question #4



*What is the dependency resolution process ?*

*TODO*

## Question #5



*What is PropertyPlaceholderConfigurer ?*

*TODO*

## Question #6



What is the concept of Reference to other beans ? And how can it be accomplished in Spring ?

*TODO*



What is the difference between ref and idref ?

*TODO*

## Question #8



What is the syntax for implementing List as a collection in the Spring Configuration ?

*TODO*

## Question #9



What is the syntax for implementing Set as a collection in the Spring Configuration ?

*TODO*

## Question #10



What is the syntax for implementing Map as a collection in the Spring Configuration ?

*TODO*

## Question #11



What is the syntax for implementing Properties as a collection in the Spring Configuration ?

*TODO*



How to define a null value to a bean property in Spring ?

`<property name="emailId" value="null" /> or  
<property name="emailId"><null /></property>`



**Student** has a property called **Department**, **Department** has a property called **Category** and **Category** has a property called **name**. Is it possible to set the value for the **name** property in the Spring Configuration ?

*<TODO>*

## Question #14



How to prevent a bean from being initialized when the container starts ?

*<TODO>*



How to refer to a Spring Configuration from another Spring Configuration file ?

*<TODO>*

## Question #16



For supporting i18n, what class we have to configure in the Spring Configuration File ?

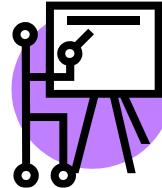
*`org.springframework.context.support.ResourceBundleMessageSource`*



## Unit 7



# Advanced Spring Bean Concepts



## Agenda

- Spring Bean Autowiring
- Bean Scopes
- Bean – Life Cycle callbacks
- Bean Inheritance
- Bean Postprocessors
- Checkpoint Summary
- Quiz

- The Spring container can **autowire** relationships between collaborating beans without using the `<constructor-arg>` and/or `<property>` tags which helps cut down on the amount of XML configuration you write for a big Spring-based application.

# Spring Bean Autowiring – Modes



Mode	Description	Example
no	No auto wiring of beans	Default
byName	Auto wire beans by property name	<code>&lt;bean autowire="byName" ... /&gt;</code>
byType	Auto wire beans by property type	<code>&lt;bean autowire="byType" ... /&gt;</code>
constructor	Auto wire beans by constructor	<code>&lt;bean autowire="constructor" ... /&gt;</code>
<del>autodetect</del>	<del>First try by constructor, then by type</del>	<del><code>&lt;bean autowire="autodetect" ... /&gt;</code></del>



Removed  
in  
Spring 3.0

- This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring.
- Example Syntax

```
<bean id="employeeBean" class="org.springbasics.Employee">  
    ...  
    ...  
</bean>
```

- Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byName* in the XML configuration file.
- It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
- Example Syntax

```
<bean id="employeeBean" class="org.springbasics.Employee"  
autowire="byName">  
    ...  
    ...  
</bean>
```

- Autowiring by property datatype.
- Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property if its type matches with exactly one of the beans name in configuration file.
- Example Syntax

```
<bean id="employeeBean" class="org.springframework.Employee"  
autowire="byType">  
    ...  
    ...  
</bean>
```
- If more than one such beans exists, a fatal exception is thrown at runtime.
  - **org.springframework.beans.factory.BeanCreationException**

- Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.
- Example Syntax

```
<bean id="employeeBean" class="org.springbasics.Employee"  
autowire="constructor">  
    ...  
    ...  
</bean>
```

- Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

Removed and no longer available from Spring 3.0 onwards !!!.

- **Excluding a bean from autowiring (p61)**

- Spring auto wiring has certain disadvantages or limitations:

Disadvantage	Description
Overriding beans configuration	If the bean configuration is specified explicitly then it overrides the bean auto wiring configuration
Unable to wire primitive types	Auto wiring is applicable only for beans and not for simple properties like primitives, String etc.
Auto wiring has confusing nature	Explicit wiring or manual wiring of beans is easier to understand than auto wiring. It is preferred to use explicit wiring if possible.
Partial wiring is not possible	Auto wiring will always try to wire all the beans through setter or constructor. It cannot be used for partial wiring.
Prevent Documentation	Wiring information may not be available to tools that may generate documentation from a Spring container.

# It's time for an exercise !!!



## *Lab 17– Autowiring*

- 🎯 Autowiring *byName*

## *Lab 18– Autowiring*

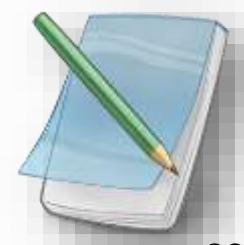
- 🎯 Autowiring *byType*

## *Lab 19– Autowiring*

- 🎯 Autowiring *by Constructor*

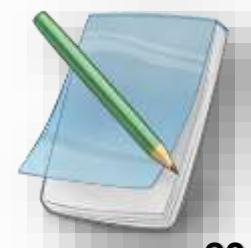
## – Steps to follow

1. Write a Address Java Bean with the following properties:  
**doorNumber, street, city, state, country, pincode.**
2. Write a Employee Java Bean with the following properties:  
**employeeName, employeeId, passportNumber, age, Address.**
3. Generate getters and setters for the properties defined above.
4. Create Spring-Config.xml and add the Employee and Address bean definitions and add the properties and set the values.
5. Instead of defining the Address Bean as a reference in the Employee Bean, use the **autowire= "byName "** tag in the Employee bean definition.
6. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext.**
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



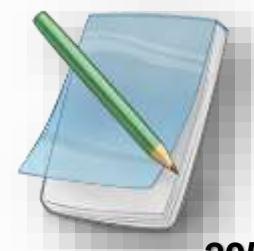
## – Steps to follow

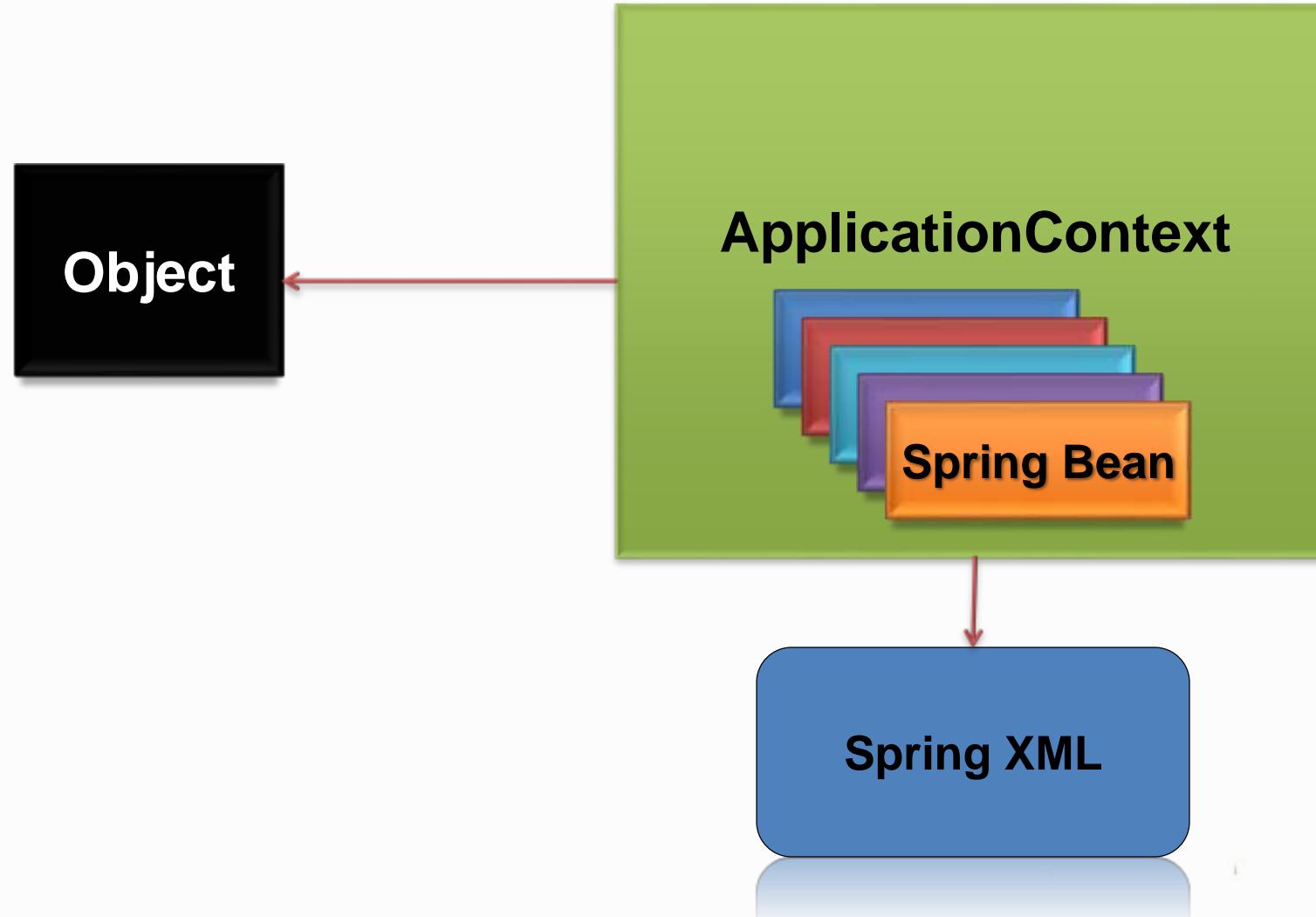
1. Copy [LAB 17 – Spring Bean Autowiring – \*byName\*](#) and create a new Java Project.
2. Instead of defining the Address Bean as a reference in the Employee Bean, use the `autowire="byType"` tag in the Employee bean definition.
3. Change the bean id of the Address Bean to some thing else, which doesn't match the Address bean property created in the Employee Bean.
4. Write Main program to read the Spring-Config.xml using `ClassPathXmlApplicationContext`.
5. Retrieve the values using the bean that is created in the previous step.
6. Run the program to display the output.



## – Steps to follow

1. Copy **LAB 17 – Spring Bean Autowiring – byName** and create a new Java Project
2. Instead of defining `<constructor-arg> <ref bean="address" />`  
`</constructor-arg>` under the Employee bean, use the  
`autowire="constructor"` tag in the Employee `<bean>` tag.
3. In the Employee Bean class, create a one argument constructor which will take Address bean.
4. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
5. Retrieve the values using the bean that is created in the previous step.
6. Run the program to display the output.





# Different types of Bean Scopes



Bean Scope	Description
singleton	Single instance of bean in every getBean() call [Default]
prototype	New instance of bean in every getBean() call
request	Single instance of bean per HTTP request
session	Single instance of bean per HTTP session
global-session	Single instance of bean per global HTTP session
thread	Single instance of bean per thread
custom	Customized scope

For this training program, we will look into 'singleton' and 'prototype' scopes only

- Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

```
<bean id="..." class="...">  
    <property name="accountDao"  
             ref="accountDao"/>  
</bean>
```

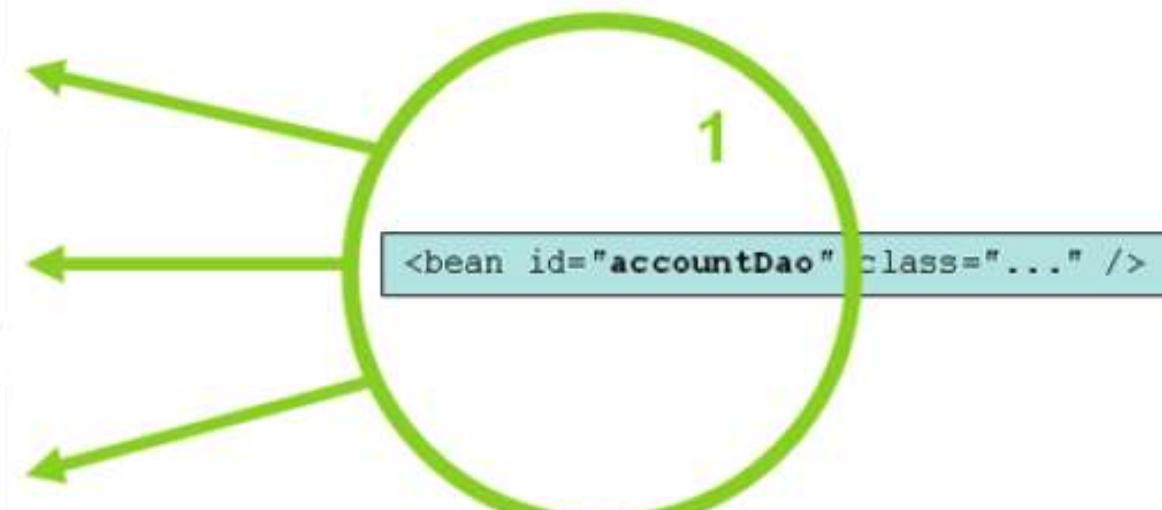
  

```
<bean id="..." class="...">  
    <property name="accountDao"  
             ref="accountDao"/>  
</bean>
```

```
<bean id="..." class="...">  
    <property name="accountDao"  
             ref="accountDao"/>  
</bean>
```

**Only one instance is ever created...**



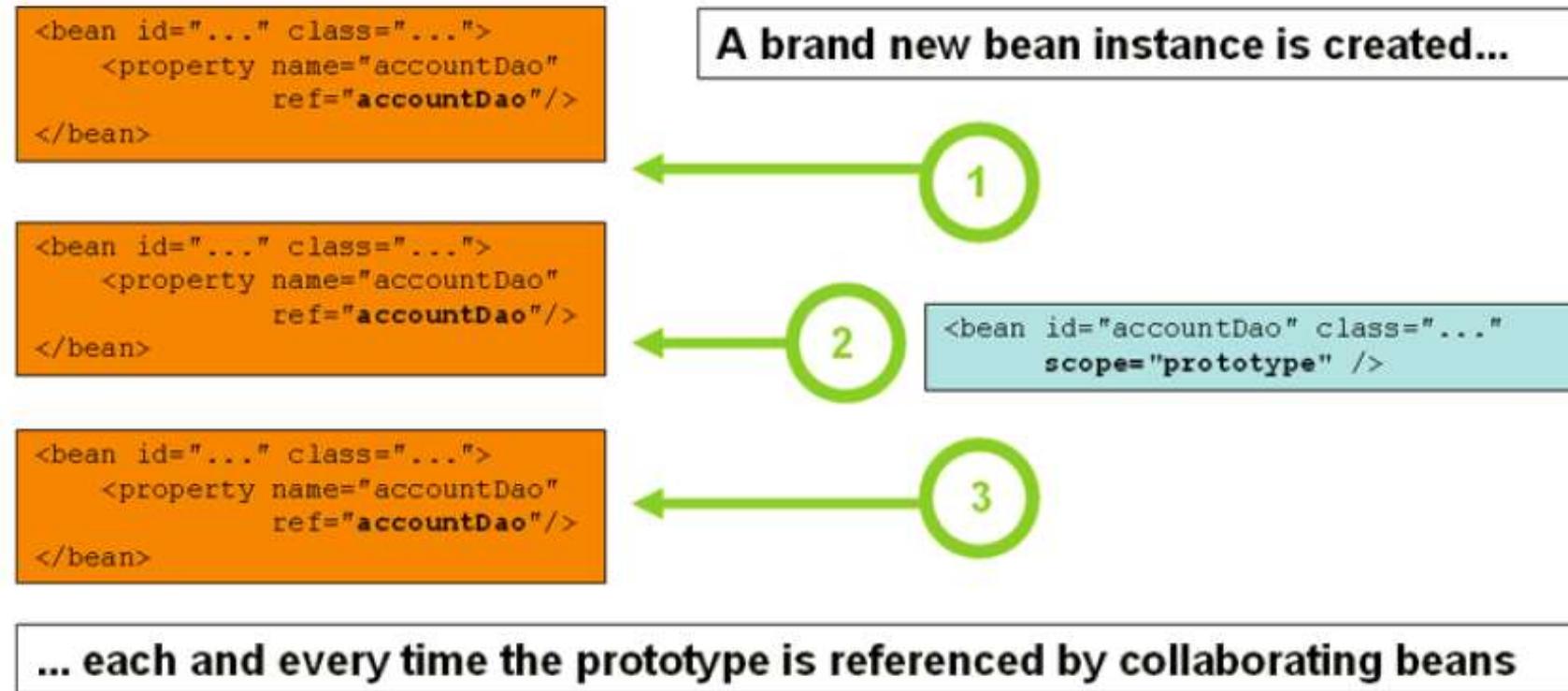
**... and this same shared instance is injected into each collaborating object**

- The singleton scope is the default scope in Spring.
- To define a bean as a singleton in XML, you would write, for example:

```
<bean id="accountService" class="com.foo.DefaultAccountService" />
<!-- the following is equivalent, though
redundant (singleton scope is the default) -->

<bean id="accountService" class="com.foo.DefaultAccountService"
scope="singleton" />
```

- Spring IoC container *creates new bean instance of the object every time a request for that specific bean is made.*
- Use the prototype scope for all state-full beans and the singleton scope for stateless beans.



- The request, session, and global session scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext).
- If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, you get an IllegalStateException complaining about an unknown bean scope.

# Singleton or Prototype: which one to choose



- The lifecycle of beans is generally managed by the Spring container. But, when a bean is a prototype, Spring does not handle the destruction of the beans. means Spring does not call the destruction callback methods of prototype beans. we must write explicit code to clean up any prototype beans.
- as Per the Spring documentation, “the Spring container’s role in regard to a prototype-scoped bean is a replacement for the Java new operator.
- All lifecycle management past that point must be handled by the client.”
- so prefer singleton unless there is a need for prototype in application.

- In a Bean Configuration File
- Using Annotations

# It's time for an exercise !!!



## *Lab 20 – Bean Scopes*

- 🎯 Singleton Bean Scope

## *Lab 21 – Bean Scopes*

- 🎯 Prototype Bean Scope

## – Steps to follow

1. Create an Employee Java Bean with the following property.  
**lastLogonAttempt**.
2. Generate getter and setter for the property defined above.
3. Create Spring-Config.xml and add Employee bean definition with two different ids: singletonScope and prototypeScope and set the **scope** attribute to singleton and prototype respectively.
4. Write Main program to read the Spring-Config.xml using  
**ClassPathXmlApplicationContext**.
5. Create 2 Singleton and 2 prototype instances.
6. Set the date for one of the Singleton and prototype to:  
`xxx.setLastLogonAttempt(new Date());`
7. Retrieve the date for both singleton objects and 1 prototype object and print them.
8. Run the program to display the output.



- From the Bean's perspective, when it got instantiated , It may be required to perform some initialization in order to get it into a usable state.
- Similarly, a cleanup might be required when the bean is about to be destroyed (to be removed from the container).
- Bean Life Cycle can be handled in the following ways:
  - Programmatically
  - XML based Configuration Metadata
  - Annotation Based

- Initialization Callbacks
  - **org.springframework.beans.factory.InitializingBean**
    - **void afterPropertiesSet() throws Exception;**
- Destruction Callbacks
  - **org.springframework.beans.factory.DisposableBean**
    - **void destroy() throws Exception;**

```
<bean id="triangle" class="org.springframeworkbasics.Triangle" >
```

```
public class Triangle implements InitializingBean,  
DisposableBean {  
{  
    public void afterPropertiesSet()  
        throws Exception {  
            // do some initialization work  
        }  
    public void destroy() throws Exception {  
        // do some destruction work  
    }  
}
```

It's time for an exercise !!!



## *Lab 22 – Bean Life Cycle Callbacks*



Programmatic Bean Life Cycle Callbacks



## – Steps to follow

1. Copy [LAB 17 – Spring Bean Autowiring – \*byName\*](#) and create a new Java Project from it. .
2. Modify the Employee Bean class so that it will implement InitializingBean, DisposableBean interfaces.
3. Implement the methods: afterPropertiesSet() and destory()
4. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
5. Retrieve the values using the bean that is created in the previous step.
6. Run the program to display the output.



- To register a shutdown hook, you call the `registerShutdownHook()` method that is declared on the **AbstractApplicationContext** class:

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class DrawingApp {
    public static void main(final String[] args) {
        AbstractApplicationContext context = new
            ClassPathXmlApplicationContext("Spring-Config.xml");
        // add a shutdown hook for the above context...
        context.registerShutdownHook();
        // app runs here.
        Triangle triangle = (Triangle) context.getBean("triangle");
        triangle.draw();
        // main method exits, hook is called prior to the app shutting down...
    }
}
```

- Declare the <bean> with the following attributes:
  - **init-method** and/or
  - **destroy-method**

## – Initialization & Destruction

```
<bean id="triangle"
      class="org.springframework.Triangle"
      init-method="myInit"
      destroy-method="cleanup">
```

```
public class Triangle{
    public void myInit() {
        // do some initialization work
    }
    public void cleanup() {
        // do some destruction work
    }
}
```

Exactly same as

```
<bean id="triangle"
      class="org.springframework.Triangle" >

public class Triangle implements InitializingBean,
DisposableBean {
    public void afterPropertiesSet()
        throws Exception {
            // do some initialization work
    }
    public void destroy() throws Exception {
        // do some destruction work
    }
}
```

- If you have too many beans having initialization and or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean. Instead framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method** attributes on the `<beans>` element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..."
       xmlns:xsi="..."
       xsi:schemaLocation="..."
       default-init-method="myInit" default-destroy-method="cleanup">
```

It's time for an exercise !!!



## *Lab 23 – Bean Life Cycle Callbacks*

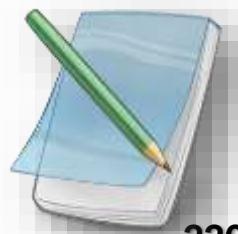


Bean Life Cycle Callbacks – using XML Configuration Metadata



## – Steps to follow

1. Copy [LAB 22 – Spring Bean Lifecycle Callbacks – Programmatically](#) and create a new Java Project from it. .
2. Modify the Employee Bean class to remove the implemented InitializingBean, DisposableBean interfaces.
3. Delete the methods: afterPropertiesSet() and destory()
4. Create your own methods called initialization() and destroy().
5. Modify Spring-Config.xml to add `init-method="initialization"` `destroy-method="destroy"` in the Employee bean definition.
6. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
7. Retrieve the values using the bean that is created in the previous step.
8. Run the program to display the output.



- Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:
  - Initialization methods are called in the following order:
    - Methods annotated with `@PostConstruct`
    - `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
    - A custom configured `init()` method
  - Destroy methods are called in the same order:
    - Methods annotated with `@PreDestroy`
    - `destroy()` as defined by the `DisposableBean` callback interface
    - A custom configured `destroy()` method



## *Lab 23-1 – Bean Life Cycle Callbacks*



### Combining Life Cycle Callbacks

- Programmatic
- XML configuration

- Declare the method with the following attributes:
  - **@PostConstruct**
  - **@PreDestroy**
- Will be covered in the next unit: Annotation-based Container Configuration

# Bean Definition Inheritance (Parent and Child Beans)



- Several individual <bean> declarations can make Spring configuration unwieldy and brittle.

```
<bean id='program001'  
      class='com.personal.program.Staff'>  
    <property name='staffId' value='program001'/>  
    <property name='name' value='Abc'/>  
    <property name='dept' value='GTB'/>  
    <property name='location' value='program 2.0' />  
</bean>
```

```
<bean id='program002'  
      class='com.personal.program.Staff'>  
    <property name='staffId' value='program002'/>  
    <property name='name' value='Xyz'/>  
    <property name='dept' value='GTB'/>  
    <property name='location' value='program 2.0' />  
</bean>
```

```
<bean id='program003'  
      class='com.personal.program.Staff'>  
    <property name='staffId' value='program003'/>  
    <property name='name' value='Pqr'/>  
    <property name='dept' value='GTB'/>  
    <property name='location' value='program 2.0' />  
</bean>
```



```
<bean id='gtb'  
      class='com.personal.program.Staff'  
      abstract='true'>  
    <property name='dept' value='GTB' />  
    <property name='location' value='program  
      2.0' />  
</bean>  
  
<bean id='program001' parent='gtb'>  
  <property name='staffId'  
            value='program001' />  
  <property name='name' value='Abc' />  
</bean>  
  
<bean id='program002' parent='gtb' >  
  <property name='staffId'  
            value='program002' />  
  <property name='name' value='Xyz' />  
</bean>  
  
<bean id='program003' parent='gtb' >  
  <property name='staffId'  
            value='program003' />  
  <property name='name' value='Pqr' />  
</bean>
```

- A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.
- A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. *Effectively, this is a form of templating.*
- When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.
- A child bean definition uses the bean class from the parent definition if none is specified, but can also override it. In the latter case, the child bean class must be compatible with the parent, that is, it must accept the parent's property values.

- Child bean definition inherits the following from the parent, with an option to add new values.
  - constructor argument values
  - Property values
  - Method overrides including Initialization Method, destroy method, static factory method
- Following settings will always taken from the Child Definition
  - depends on
  - autowire mode
  - dependency check
  - Singleton
  - Scope
  - lazy init

# It's time for an exercise !!!



## *Lab 24 – Bean Inheritance*

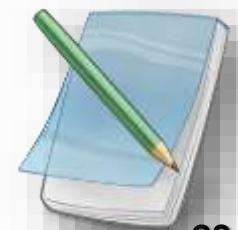


Implement Bean Definition Inheritance



## – Steps to follow

1. Copy [LAB 17 – Spring Bean Autowiring – \*byName\*](#) and create a new Java Project from it.
2. Update the Spring-Config.xml: Create 2 new beans of type Employee Bean and set the **parent** attribute to Employee Bean id created in the previous step.
3. Change the values of the properties defined in the newly defined beans.
4. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
5. Retrieve the values using the bean that is created in the previous step.
6. Run the program to display the output.



- The `BeanPostProcessor` interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth.
- If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more `BeanPostProcessor` implementations.
- You can configure multiple `BeanPostProcessor` instances, and you can control the order in which these `BeanPostProcessors` execute by setting the `order` property. You can set this property only if the `BeanPostProcessor` implements the `Ordered` interface;

- `org.springframework.beans.factory.config.BeanPostProcessor`
- `public Object postProcessBeforeInitialization(final Object bean, final String beanName) throws BeansException`
- `public Object postProcessAfterInitialization(final Object bean, final String beanName) throws BeansException`

# Bean Post Processors – Example



```
public class HelloWorld {  
    private String message;  
    public void setMessage(String message){  
        this.message = message;  
    }  
    public void getMessage(){  
        System.out.println("Your Message : " +  
            message);  
    }  
    public void init(){  
        System.out.println("Bean is going  
            through init.");  
    }  
    public void destroy(){  
        System.out.println("Bean will destroy  
            now.");  
    }  
}
```

```
public class InitHelloWorld implements BeanPostProcessor {  
    public Object postProcessBeforeInitialization(Object bean,  
                                                String beanName) throws BeansException {  
        System.out.println("BeforeInitialization : " + beanName);  
        return bean; // you can return any other object as well  
    }  
    public Object postProcessAfterInitialization(Object bean,  
                                                String beanName) throws BeansException {  
        System.out.println("AfterInitialization : " + beanName);  
        return bean; // you can return any other object as well  
    }  
  
<bean id="helloWorld" class="com.HelloWorld"  
      init-method="init" destroy-method="destroy">  
  <property name="message" value="Hello World!" />  
</bean>  
  
<bean class="com.InitHelloWorld" />
```

It's time for an exercise !!!



## *Lab 25— Spring Bean Post Processing*

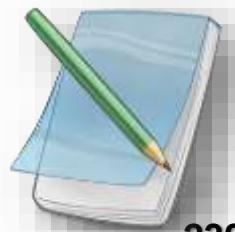


Implement Bean Post Processor



## – Steps to follow

1. Copy [LAB 23 – Spring Bean Lifecycle Callbacks](#) – using XML config and create a new Java Project from it.
2. Create **BeanPostProcessorImplementation** class which implements **BeanPostProcessor** interface.
3. Implement: **postProcessBeforeInitialization** and **postProcessAfterInitialization** methods.
4. Define the **BeanPostProcessorImplementation** in the Spring-Config.xml.
5. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



- Sometimes it is required that our beans needs to get some information about **Spring container** and its **resources**.
- For example, sometime bean need to know the current Application Context using which it can perform some operations like loading specific bean from the container in a programmatic way.
- So to make the beans aware about this, spring provides lot of Aware interfaces.
- All we have to do is, make our bean to implement the Aware interface and implement the setter method of it.
- `org.springframework.beans.factory.Aware` is the root marker interface.
- All of the Aware interfaces which we use are the sub interfaces of the Aware interface.

## Some of the commonly used Aware Interfaces are:

ApplicationContextAware	Bean implementing this interface can get the current application context and this can be used to call any service from the application context
BeanFactoryAware	Bean implementing this interface can get the current bean factory and this can be used to call any service from the bean factory
BeanNameAware	Bean implementing this interface can get its name defined in the Spring container.
MessageSourceAware	Bean implementing this interface can get the access to message source object which is used to achieve internationalization
ServletContextAware	Bean implementing this interface can get the access to ServletContext which is used to access servlet context parameters and attributes

## Some of the commonly used Aware Interfaces are:

ServletConfigAware

Bean implementing this interface can get the access to ServletConfig object which is used to get the servlet config parameters

ApplicationEventPublisherAware

Bean implementing this interface can publish the application events and we need to create a listener which listen this event.

ResourceLoaderAware

Bean implementing this interface can load the resources from the classpath or any external file.



## *Lab 25-1 – Spring Aware interfaces for beans*

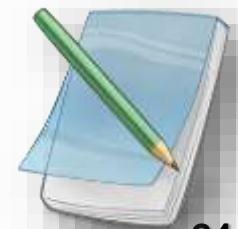


Implement the following interfaces

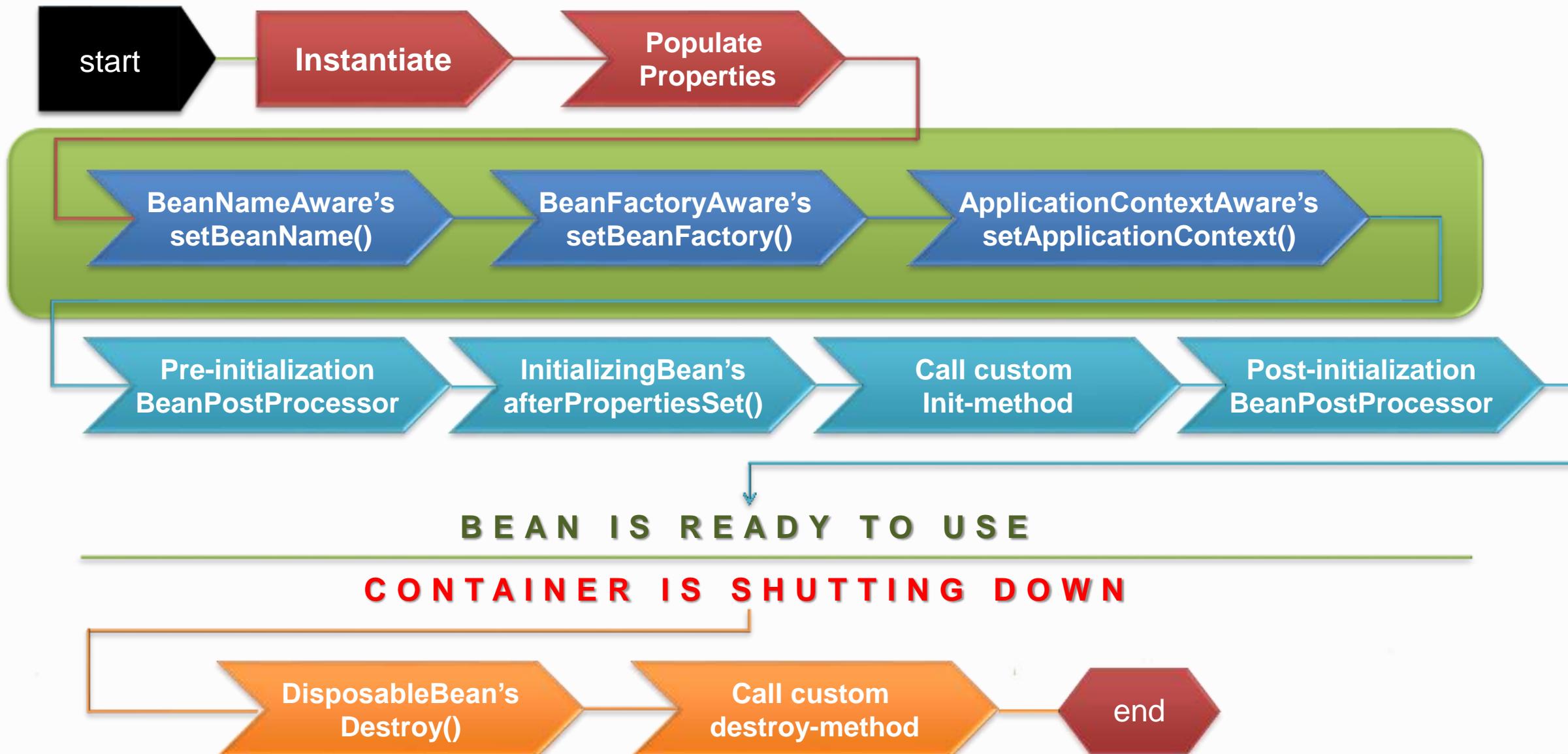
- BeanNameAware
- BeanFactoryAware
- ApplicationContextAware

## – Steps to follow

1. .



# Spring Bean Life Cycle – Summary





*Following concepts have been covered so far...*

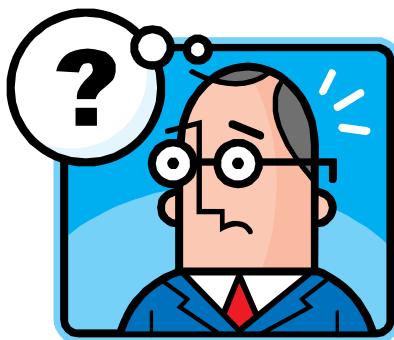
- Spring Bean Autowiring
- Bean Scopes
- Bean – Life Cycle callbacks
- Bean Definition Inheritance
- Bean Postprocessors
- Spring Aware interfaces for beans



QUESTION



## TEST YOUR UNDERSTANDING



## Question #1



*Describe Autowiring & Different ways of Autowiring ?*

<<??>>

## Question #2



*What are the rules for Autowiring byName ?*

<<??>>

## Question #3



*What are the rules for Autowiring byType?*

<<??>>

## Question #4



*What are the rules for Autowiring by Constructor?*

<<??>>

## Question #5



**TRUE**  
**FALSE**

Autowiring is best suitable over manual wiring of beans

*False*

## Question #6



*What are the Limitations of Autowiring ?*

<<??>>

## Question #7



*How can you define the scope of a Bean in Spring ?*

<<??>>

## Question #8



*What are different Bean Scopes ?*

<<??>>

## Question #9



*Singleton vs Prototype scoped Bean ?*

<<??>>

## Question #10



What are the different ways of Customizing the Lifecycle  
Callbacks of a bean ?

<<??>>

## Question #11



How can you configure the Bean Lifecycle Callbacks  
Programmatically ?

<<??>>

## Question #12



How can you configure the Bean Lifecycle Callbacks using XML Configuration Metadata ?

<<??>>

## Question #13



If you have too many beans having initialization and or destroy methods with the same name, are you going with declaring the init-method and destroy-method on each individual bean ? If not, what is the procedure ?

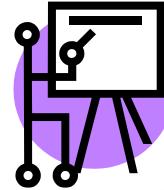
<<??>>



## Unit 8



# Annotation based Spring Configuration



## Agenda

- Annotation Based Spring Configuration
- Spring Annotations
- JSR 250 Annotations
- Java Based Spring Configuration
- Checkpoint Summary
- Quiz

- Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written.
- Following three important methods to provide configuration metadata to the Spring Container are:
  - *XML Based Configuration File*
  - **Annotation based Configuration**
  - *Java based configuration*

- Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
- Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.
- Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.

Following annotations are being covered for this Training Programme.

- @Required
- @Autowired
- @Scope
- @Qualifier
- @Component
- @Bean

A few JSR 250 Annotations

- @Resource (Autowiring by name)
- @PostConstruct
- @PreDestroy

- **@Required**
  - The `@Required` annotation applies to bean property setter methods.
  - It indicates that the affected bean property must be populated in XML configuration file at configuration time otherwise the container throws a **BeanInitializationException** exception.

## *Lab 26 – Spring Configuration Metadata using Annotations*



- Write a program to use the @Required Annotation



## – Steps to follow

1. Create a Country Java Bean and set two properties countryName and capital
2. Create Capital Java Bean and set two properties capitalName and capitalPinCode.
3. For the Country Java Bean's setCapital() method, append **@Required** annotation.
4. Create the Spring configuration XML and configure both Country and Capital beans.
5. Write Main program to read the Spring-Config.xml using **ClassPathXmlApplicationContext**.
6. Retrieve the values using the bean that is created in the previous step.
7. Run the program to display the output.



- **@Autowired**
  - Used to inject dependency automatically.
  - Provides more fine-grained control over where and how autowiring should be accomplished.
  - Can be used to autowire bean on the setter method just like **@Required** annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.
  - Use **@Autowired** annotation on setter methods to get rid of the **<property>** element in XML configuration file.
  - When Spring finds an **@Autowired** annotation used with setter methods, it tries to perform **byType** autowiring on the method.

- **@Autowired**
  - By default, the `@Autowired` annotation implies the dependency is required similar to `@Required` annotation, however, you can turn off the default behavior by using (`required=false`) option with `@Autowired`.
    - **`@Autowired(required=false)`**

## *Lab 27 – Spring Configuration Metadata using Annotations*



Write a program to use the @Autowired Annotation



- Steps to follow



- **@Qualifier**
  - There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property,
  - in such case you can use `@Qualifier` annotation along with `@Autowired` to remove the confusion by specifying which exact bean will be wired

- **@Qualifier**
  - There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property,
  - in such case you can use `@Qualifier` annotation along with `@Autowired` to remove the confusion by specifying which exact bean will be wired

## *Lab 28 – Spring Configuration Metadata using Annotations*



- Write a program to use the @Qualifier Annotation



- Steps to follow



# Enabling Spring Annotation



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/bean
       Other namespace declarations follows... >
```

```
<!-- For @Required Annotation -->
<bean class="org.springframework.beans.factory.annotation.
RequiredAnnotationBeanPostProcessor" />
<!-- For @Autowired Annotation -->
<bean class="org.springframework.beans.factory.annotation.
AutowiredAnnotationBeanPostProcessor"/>
<!-- and so on... -->
```

Option 1

```
<!--you can have a one line context annotation config as below. -->
<context:annotation-config/>
```

Option 2

```
</beans>
```

- @Resource (Autowiring by name)
- @PostConstruct
- @PreDestroy

- @PostConstruct, @PreDestroy
  - To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters.
  - The **init-method** attribute specifies a method that is to be called on the bean immediately upon instantiation.
  - Similarly, **destroy-method** specifies a method that is called just before a bean is removed from the container.
  - You can use `@PostConstruct` annotation as an alternate of initialization callback and `@PreDestroy` annotation as an alternate of destruction callback



## *Lab 29 – Spring Configuration Metadata using Annotations*



Write a program to use the following JSR 250 Annotations

`@PostConstruct`

`@PreDestroy`

- Steps to follow



- @Resource
  - The `@Resource` annotation takes a 'name' attribute which will be interpreted as the bean name to be injected.
  - @Resource will follow **byName** auto-wiring semantics. If no match found, **byType** will be applied.
  - @Resource is similar to @Autowired in functionality.
  - @Autowired is Spring's Annotation(only usable in Spring) whereas @Resource is JSR-250 Common Annotation equivalent (works the same in Java EE5).
  - Though the functionality for the two annotations being same, @Autowired is the only annotation that you can put on constructors.



## *Lab 30 – Spring Configuration Metadata using Annotations*



Write a program to use the JSR 250 - @Resource Annotation

- Steps to follow



- Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written.
- Following three important methods to provide configuration metadata to the Spring Container are:
  - *XML Based Configuration File*
  - *Annotation based Configuration*
  - **Java based configuration**

- So far we saw how to configure Spring beans using XML configuration file.
- It is really not required to learn how to proceed with Java based configuration if you are comfortable with XML based configuration because you are going to achieve the same result using either of the configurations available.
- Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.
- Since Spring 3, Java Configuration features are included in core Spring module, to allow developer to move bean definition and Spring configuration out of XML file into Java class.
  - @Configuration
  - @Bean

- @Configuration and @Bean Annotations
  - Annotating a class with the `@Configuration` indicates that the class can be used by the Spring IoC container as a source of bean definitions.
  - The `@Bean` annotation tells Spring that a method annotated with `@Bean` will return an object that should be registered as a bean in the Spring application context.

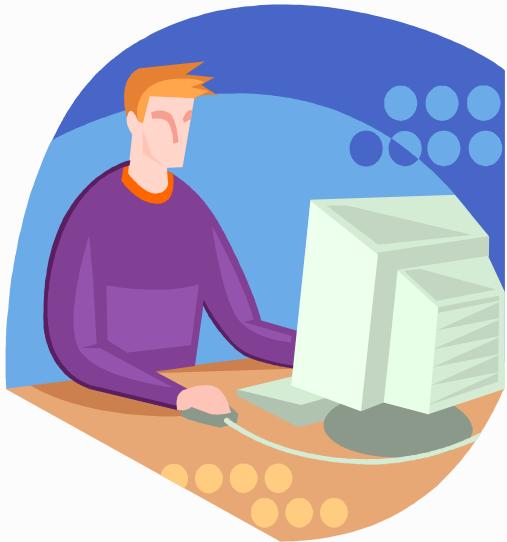
It's time for an exercise !!!



## *Lab 31 – Spring Configuration Metadata using Java*



Write a program to use @Configuration, @Bean Annotations



## – Steps to follow





*Following concepts have been covered so far...*

- Annotation Based Spring Configuration
- Spring Annotations
- JSR 250 Annotations
- Java Based Spring Configuration



QUESTION



## TEST YOUR UNDERSTANDING





*List out the Annotations covered so far ?*

`@Required`  
`@Autowired`  
`@Qualifier`  
`@PostConstruct`  
`@PreDestroy`  
`@Resource`  
`@Configuration`  
`@Bean`

## Question #2



*List out the JSR 250 Annotations covered so far ?*

`@PostConstruct`  
`@PreDestroy`  
`@Resource`

## Question #3



*What does @Required Annotation does ?*

*<TODO>*

## Question #4



*What does @Autowired & @Qualifier Annotations do ?*

*<TODO>*



*@Autowired vs @Resource Annotations*

*<TODO>*

## Question #6



*What @PostConstruct and @PreDestroy annotations do ?*

*<TODO>*

## Question #7



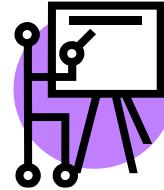
*How can you achieve the Java Based Spring configuration ?*

*@Configuration*

*@Bean*



## Unit 9



### Agenda

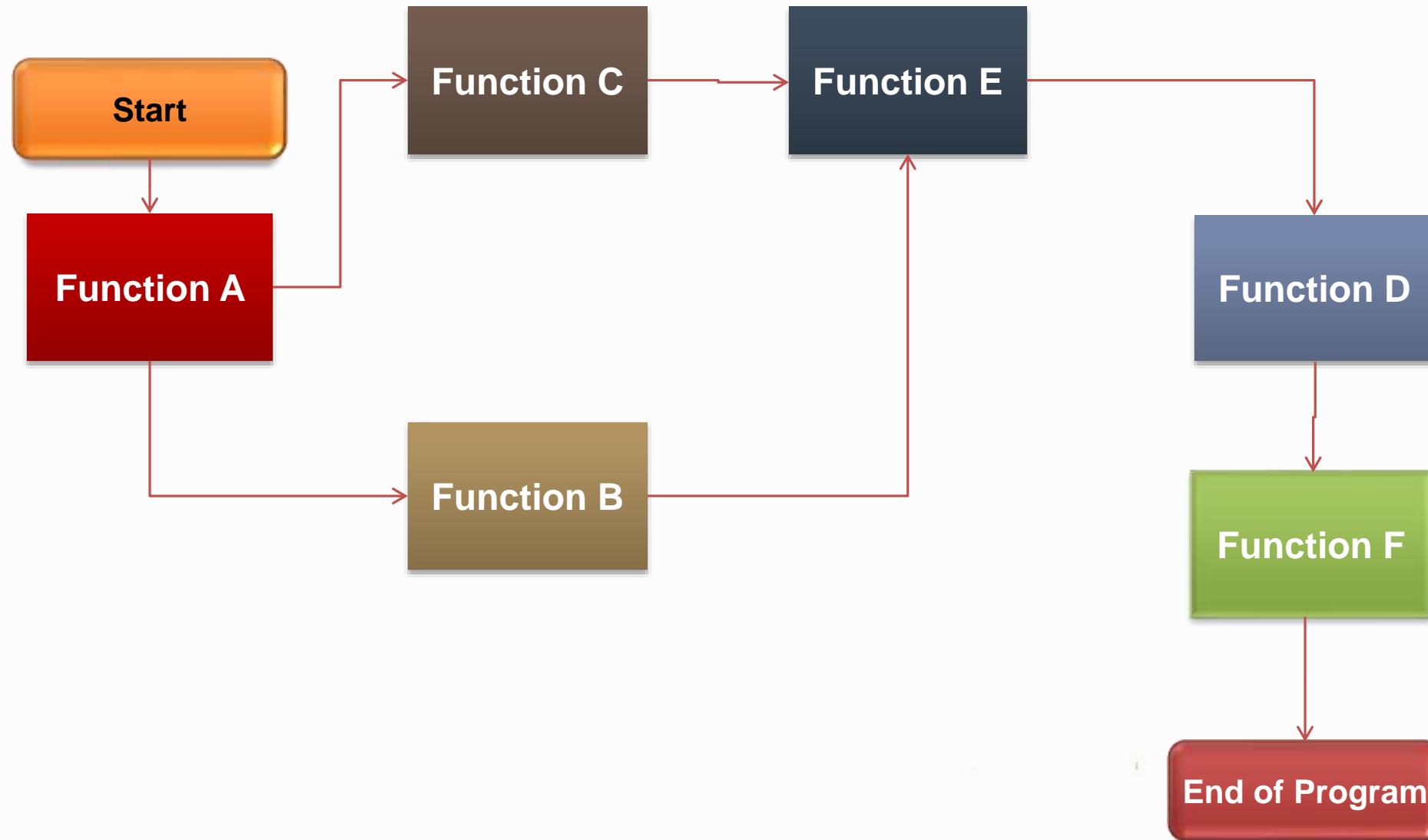
- Evolution of the Programming
- The *cross-cutting* concerns
- Evolution of AOP
- Setting up AOP Environment
- AOP Concepts
- Checkpoint Summary
- Quiz



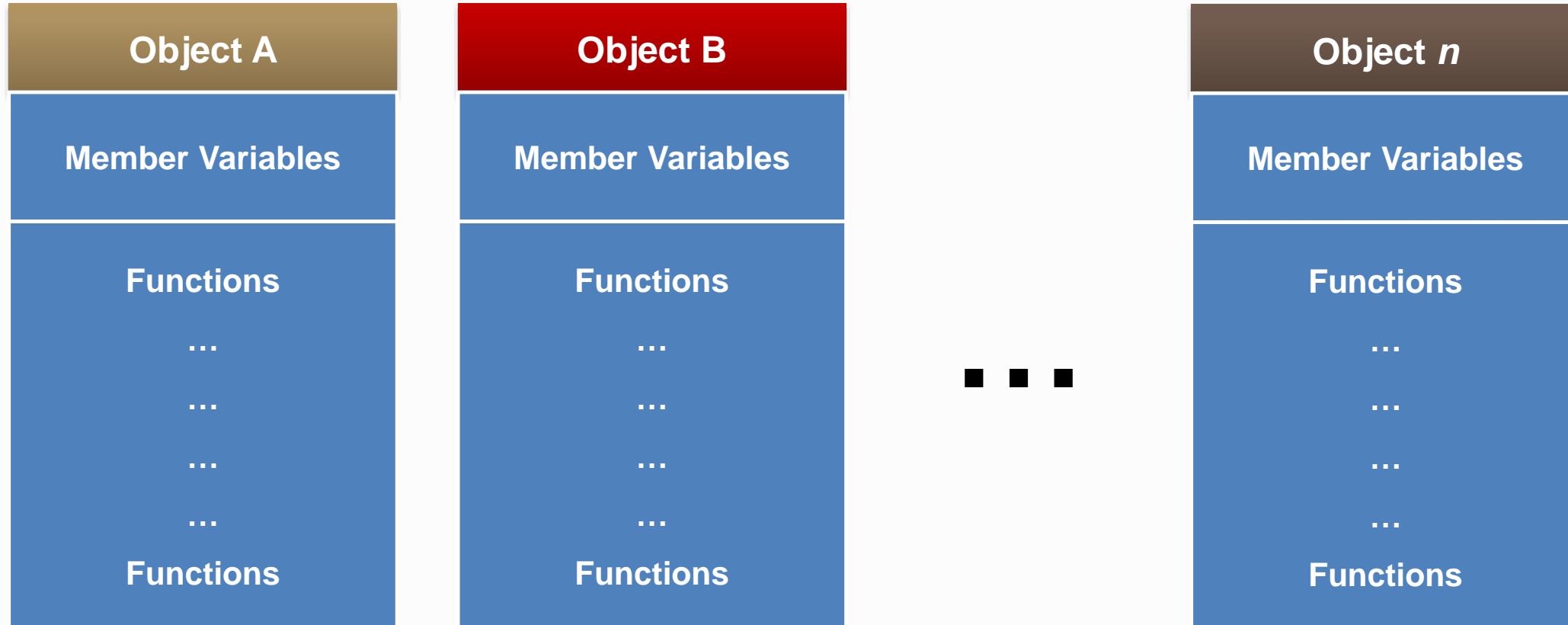
**Modular programming** is a software design technique that emphasizes separating the functionality of a **program** into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

A large, red, three-dimensional style arrow pointing to the right, containing the text "Functional Programming".A large, dark grey, three-dimensional style arrow pointing to the right, containing the text "Object-Oriented Programming".A large, gold, three-dimensional style arrow pointing to the right, containing the text "Aspect Oriented Programming".

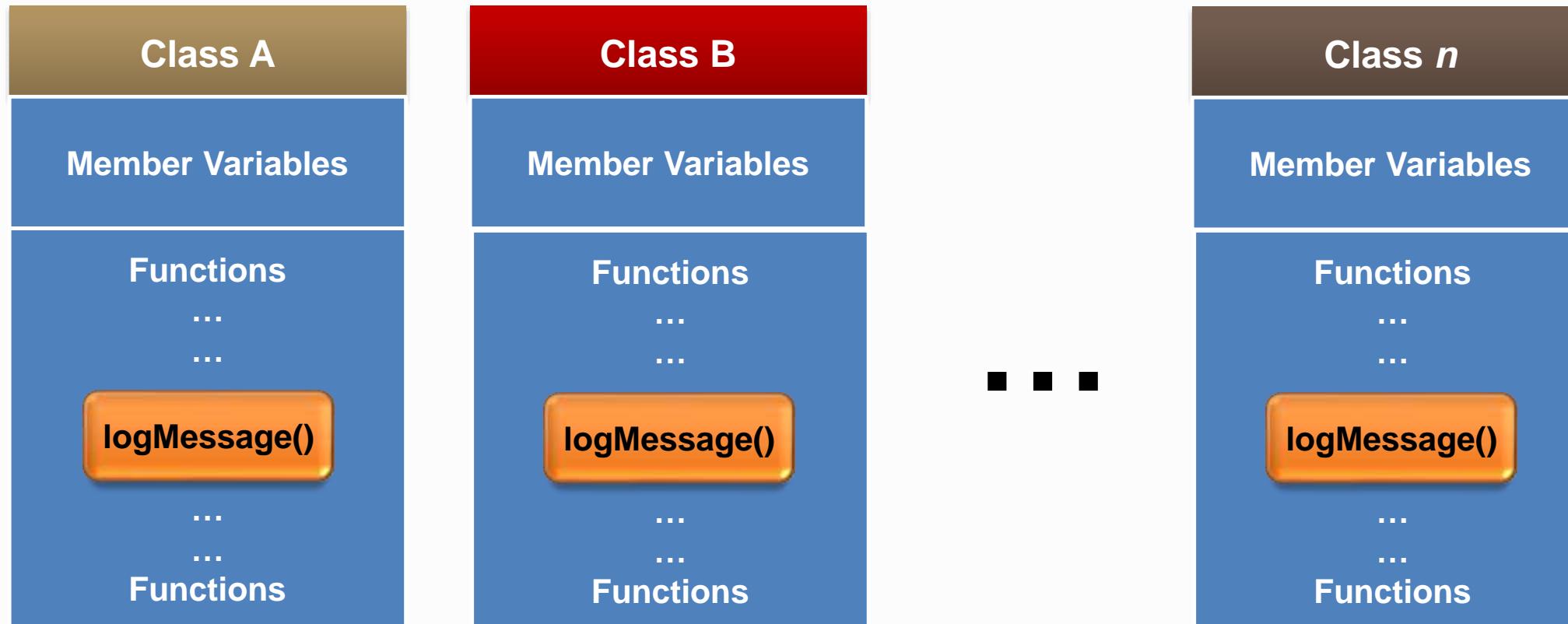
# Functional Programming



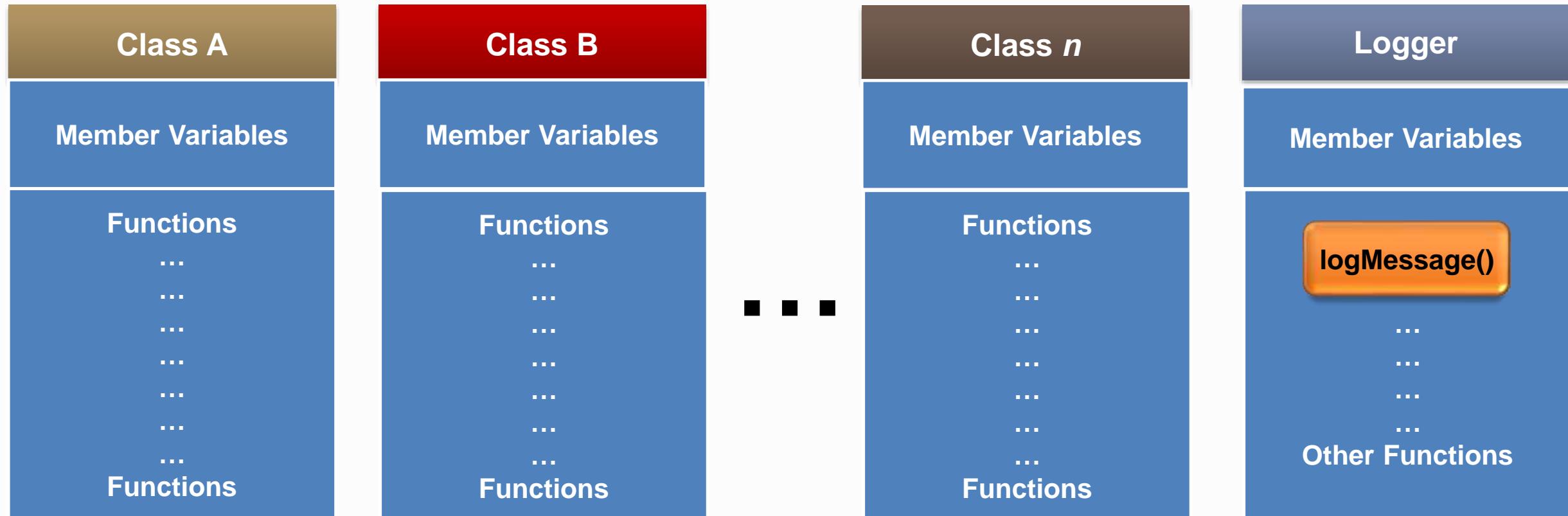
# Object Oriented Programming



# Object Oriented Programming *contd...*



# Object Oriented Programming *contd...*



- Applications must be concerned about the General Functionality that is required in many parts of your application are:
  - Logging
  - Tracing
  - Transaction Management
  - Security
  - Caching
  - Error Handling
  - Performance Monitoring
  - Custom Business Rules
- These concerns often find their way in to the application components whose core responsibility is something else.

- Failing to Modularize the Cross Cutting concerns lead to two problems
  - **Code Tangling**
    - A Coupling of concerns
  - **Code Scattering**
    - The same concern spread across modules

## – Code Tangling

```
public class RewardNetworkImpl implements RewardNetwork {  
    public RewardConfirmation rewardAccountFor(Dining dining) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        Account a = accountRepository.findByCreditCard(...);  
        Restaurant r = restaurantRepository.findByMerchantNumber(...);  
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);  
        ...  
    }  
}
```

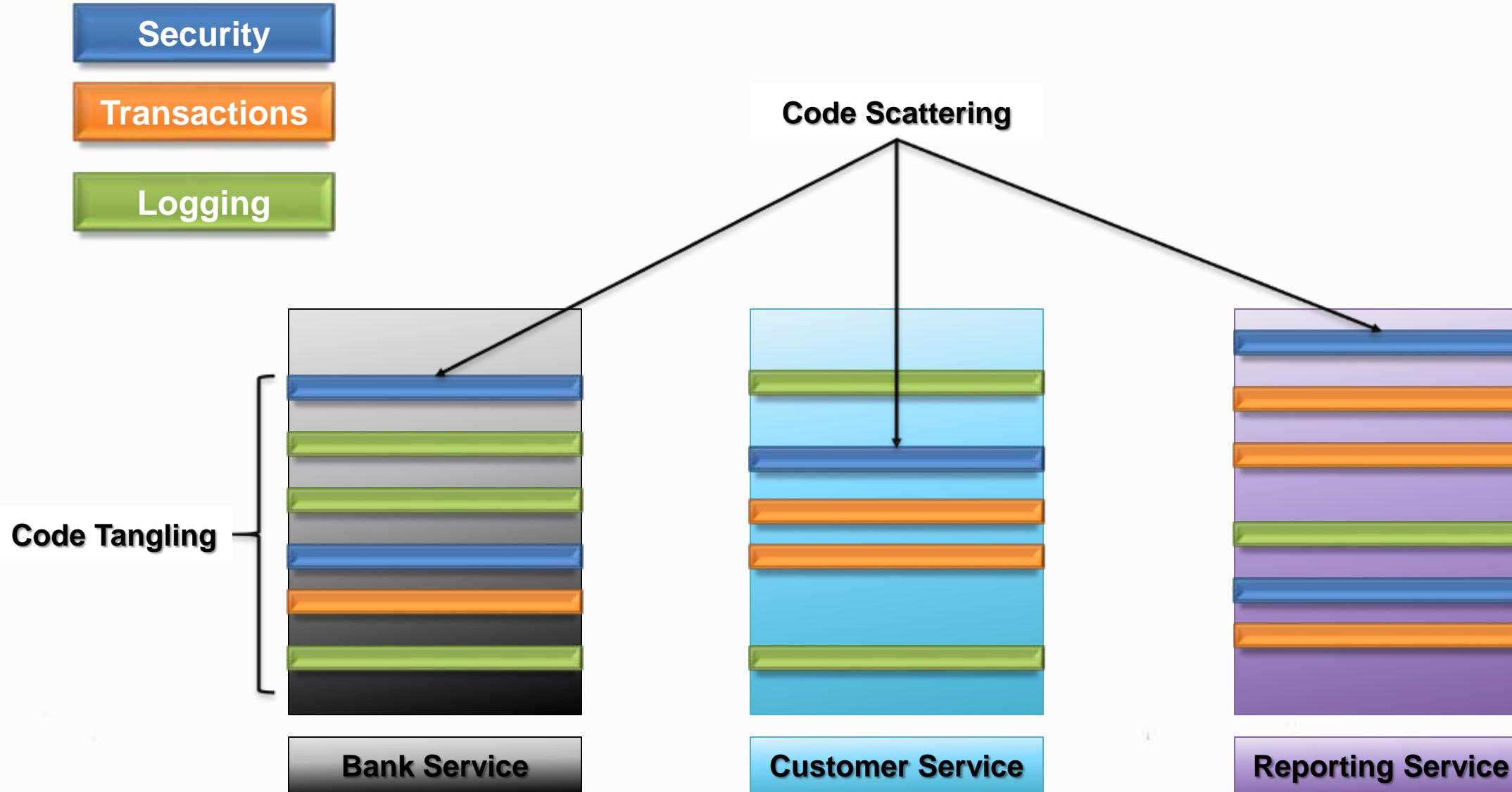
Mixing of concerns

- Code Scattering

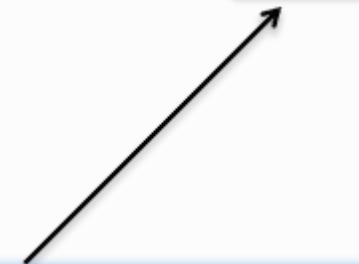
```
public class HibernateAccountManager implements AccountManager {  
    public Account getAccountForEditing(Long id) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }
```

```
public class HibernateMerchantReportingService implements  
MerchantReportingService {  
    public List<DiningSummary> findDinings(String merchantNumber,  
                                              DateInterval interval) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }
```

# System Evolution without Modularization



- Perform a role-based security check before every application method



- Problems we see in the previous diagram includes:
  - Too many relationships to the cross-cutting objects resulting in littering of the code that isn't aligned with the core functionality. (**Poor Traceability**)
  - Code is still required in all methods resulting in code duplication. (**Less Code Reuse**)
  - You can not change the code all at once resulting in less *durability, maintainability, Productivity*.

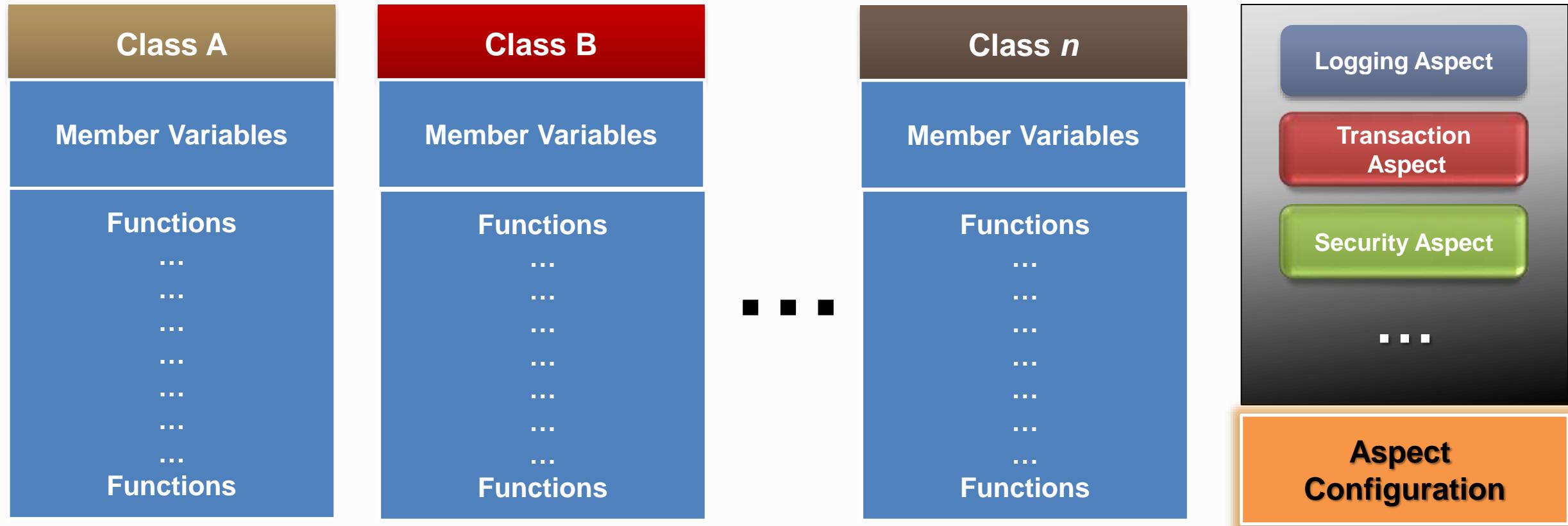
“AOP is a programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns”

– *Wikipedia*

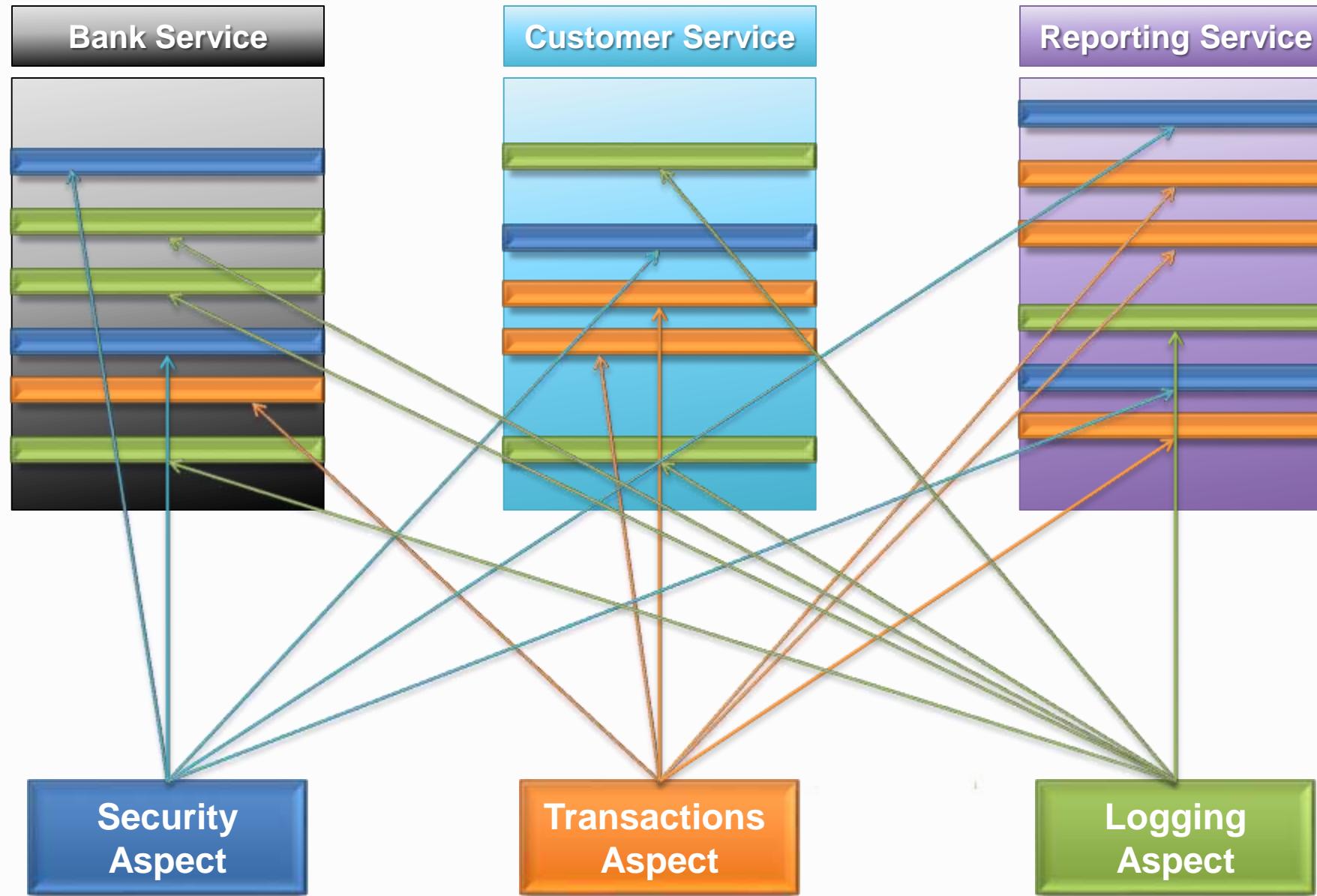
- Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.
- Not a feature of Spring Framework but is a model of Programming.
- Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect.
- A programming technique that promotes separation of concerns by modularizing these services and then apply them declaratively to the components that they should affect.
- AoP ensure that POJOs remain plain.

- AOP enables Modularization of cross-cutting concerns.
  - To avoid tangling
  - To eliminate scattering.

- Implement your Main Application logic
  - *by focusing on the main problem.*
- Write aspect to implement your cross cutting concerns
  - *Spring Framework provides various aspects out-of-the-box.*
- Weave the aspects into your application
  - *Adding the cross-cutting behaviors to the right places.*

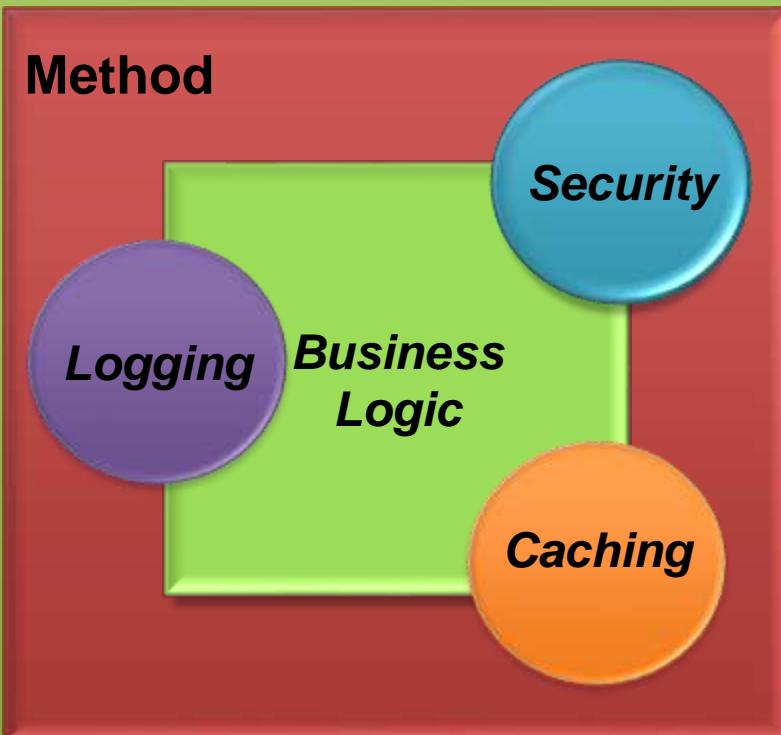


# System Evolution with Modularization



## *Standard OOP Implementation*

Object



## *AOP Implementation*

Security

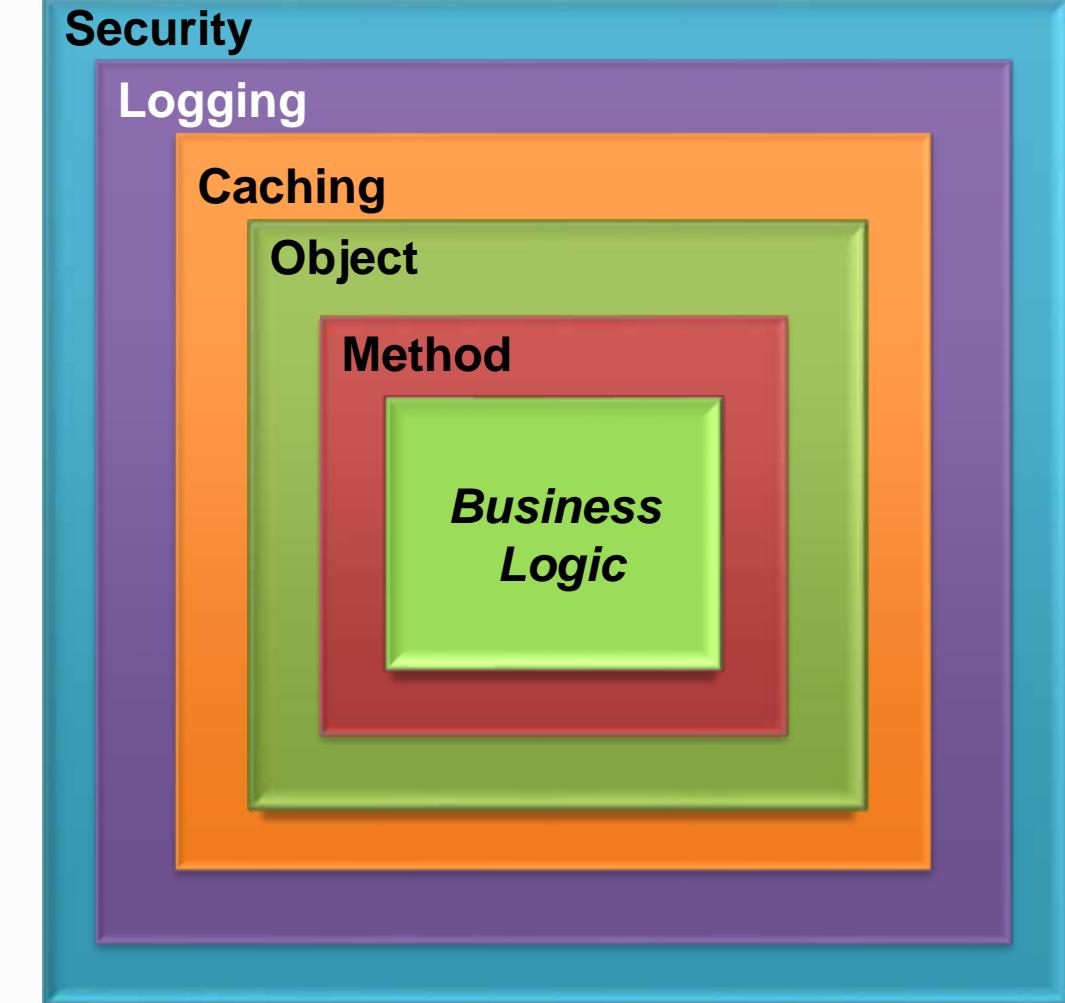
Logging

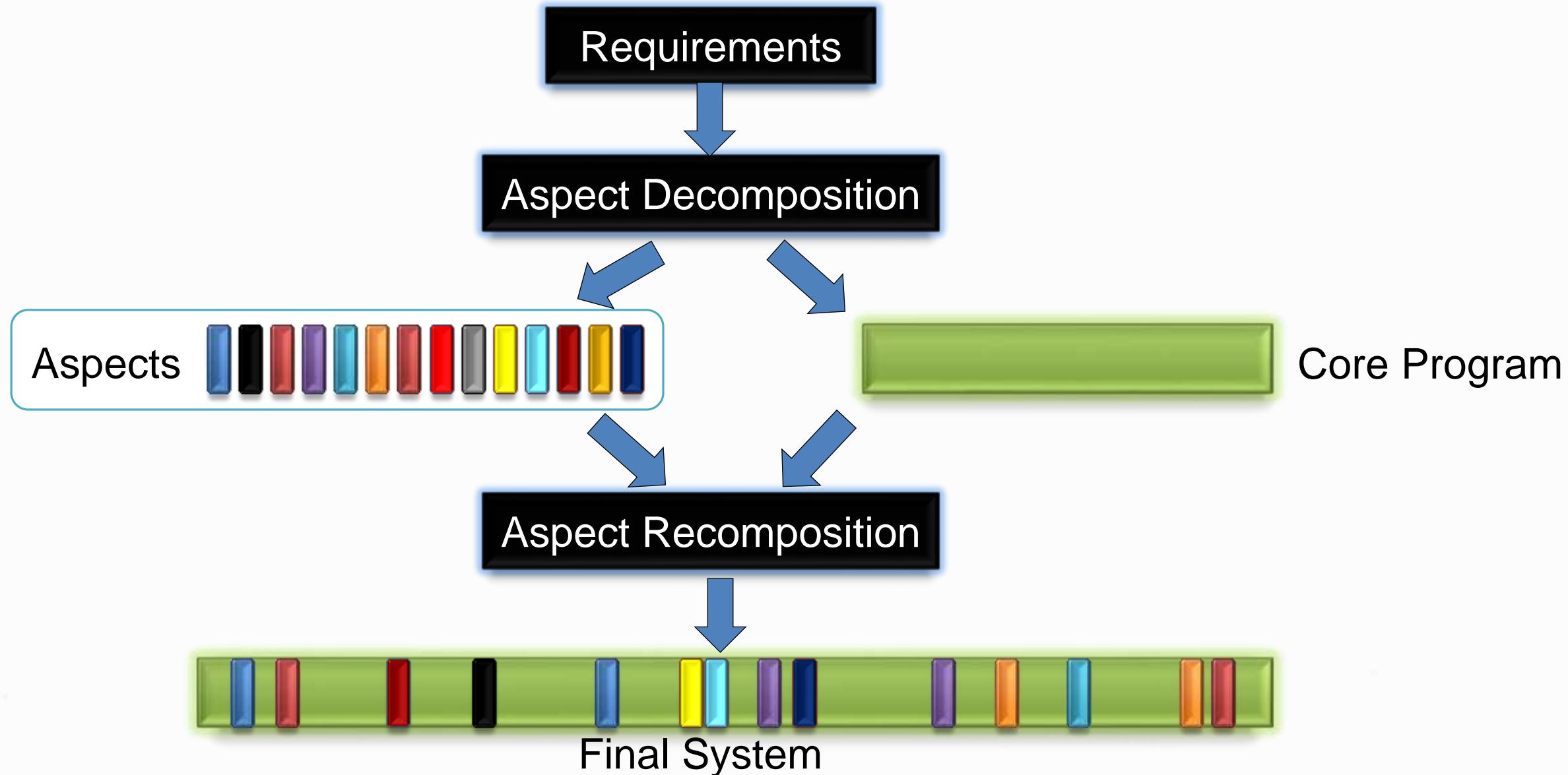
Caching

Object

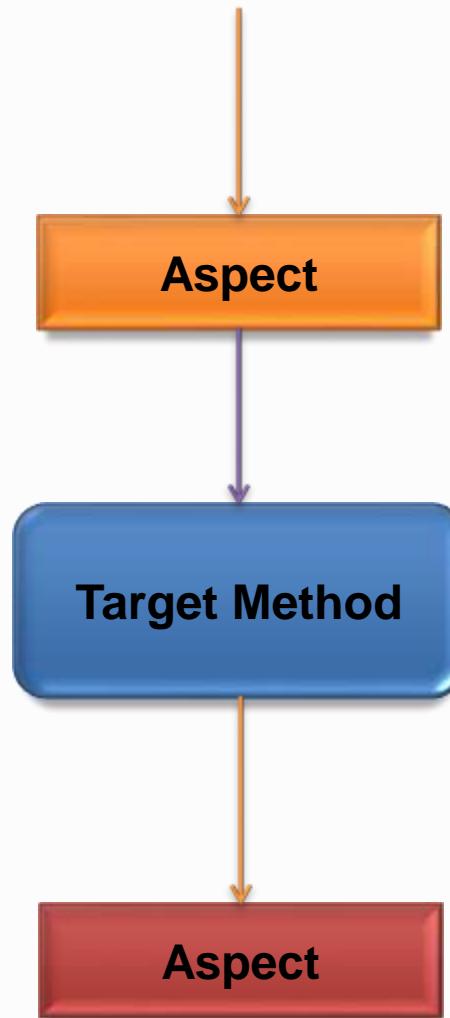
Method

Business  
Logic





# Wrapping Aspects around methods



- AspectJ
  - Original AOP Technology ( First Version in 1995)
  - Offers a full AOP Language
- Spring AOP
  - Java based AOP framework with AspectJ integration.
  - Focuses on using AOP to solve the Enterprise Problems.

# Setting up AOP Environment



- Download the following Jars for running the AOP Samples..

# It's time for an exercise !!!



## *Lab 32-42 – AOP Jump Start*



Write programs that covers the AOP concepts



Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Aspect

a modularization of a concern that cuts across multiple classes.  
Transaction management is a good example of a crosscutting concern in enterprise Java applications.

In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the `@Aspect` annotation (the `@AspectJ` style).

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Join Point

a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Advice

Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

## Before Advice

Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

## After Returning Advice

Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

## After Throwing Advice

Advice to be executed if a method exits by throwing an exception.

## After (finally) Advice

Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

## Around Advice

Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Pointcut

A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Introduction

Declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Target Object

Object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## AOP Proxy

an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Aspect

Join point

Advice

Pointcut

Introduction

Target object

AOP Proxy

Weaving

## Weaving

Linking aspects with other application types or objects to create an advised object.

This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime.

Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.



*Following concepts have been covered so far...*

- Evolution of the Programming
- Functional Programming
- Object Oriented Programming
- The *cross-cutting* concerns
- Evolution of AOP
- Setting up AOP Environment
- AOP Concepts



QUESTION



## TEST YOUR UNDERSTANDING



## Question #5



Core container has AOP as one of its module.  
best suitable over manual wiring of beans

**TRUE**

**FALSE**

*False*

*AOP is not the part of spring core container.*

## Question #1



*What are cross cutting concerns ?*

*<<TODO>>*

## Question #1



*What is Modular Programming ?*

*<<TODO>>*

## Question #1



What are different types of Advices in Spring AOP ?

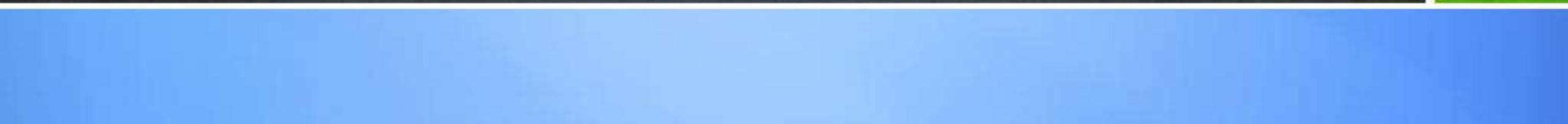
*<<TODO>>*

## Question #1



List out different Annotations used in AOP ?

*<<TODO>>*





## Unit 10



**Best  
Practices**

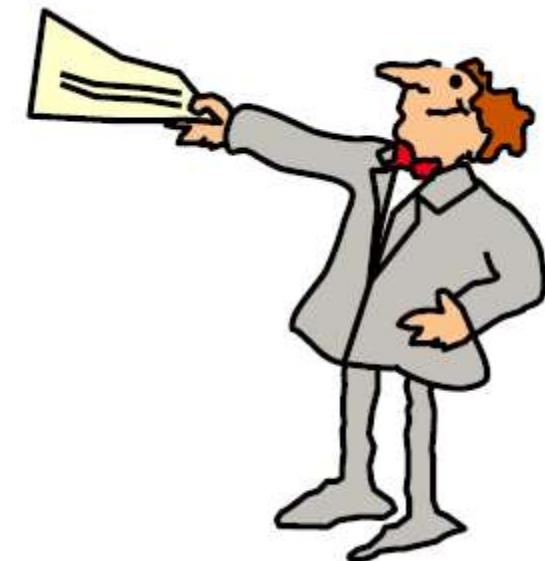


**Agenda**



**Best Practices**

- Avoid Autowiring.
- Use XML or Annotation based Configuration to prevent coupling.
- Prefer setter injection over constructor injection
- Prefer type over index for constructor argument matching
- Reuse bean definitions as much as possible
- Always use ids as bean identifiers
- Always externalize properties
- Use dependency-check at the development phase





QUESTION

## Further References *for further knowledge*

- Spring Source Reference Documentation
- Safari Books Online Account, Various Videos found in YouTube.
- Books including: Spring In Action





# Thank you

