Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Course Outline

- Lesson 1: What is Spring Framework?
- Lesson 2: Spring Framework History.
- Lesson 3: Spring Framework introduction (Putting a Spring into Hello world)
- Lesson 4: Spring Modules
- Lesson 5: Core Container (Bean Overview)
- Lesson 6: Core Container (Dependancies)
- Lesson 7: Core Container (Bean Scopes)

Course Outline (Ex.)

- Lesson 8: Core Container (Customizing the Nature of a Bean)
- Lesson 9: Core Container (Annotation-based Configuration)
- Lesson 10: Core Container (Classpath Scanning and Managed Components)
- Lesson 11: Core Container (Using JSR 330 Standard Annotations)
- Lesson 12: Core Container (Java-based Container Configuration)
- *** **References & Recommended Reading**

Lesson 1

What is Spring Framework?

What is Spring Framework?

- What We Mean by "Spring"?
  - The term "Spring" means different things in different contexts.
- Spring makes it easy to create Java enterprise applications.
- It provides everything (Core, Testing, Data Access, Web Servlets, Web Reactive, Remoting, JMS, JCA, JMX, Email, Tasks, Scheduling, Cache) you need to embrace the Java language in an enterprise environment.
- Support for Groovy and Kotlin as alternative languages on the JVM.
- Flexibility to create many kinds of architectures depending on an application's needs, including messaging, transactional data, persistence, and web..
- Spring is open source.

What is Spring Framework?

- Spring is, in fact, complementary to Java EE.
- It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases. This has helped Spring to successfully evolve over a very long time.
- The Spring Framework is divided into modules.
  - You must use the core container module, including a configuration model and a dependency injection mechanism.
  - Applications can choose which modules they need.
- It also includes the Servlet-based Spring MVC web framework and, in parallel, the Spring WebFlux reactive web framework.

What is Spring Framework?

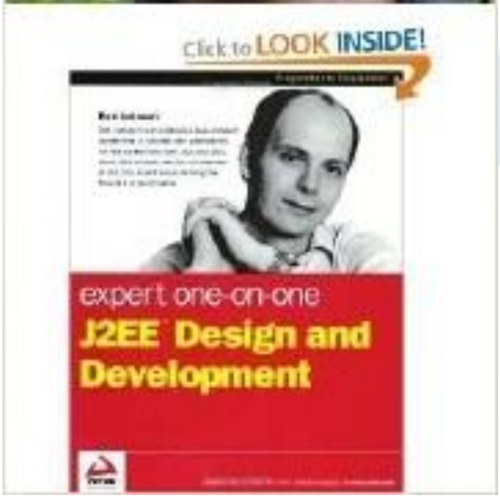
- Different packaging technique:
  - In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control.
  - Others may run as a single jar with the server embedded, possibly in a cloud environment.
  - Others may be standalone applications (such as batch or integration workloads) that do not need a server.

Lesson 2

Spring Framework History

Spring Releases Roadmap

- The first version was written by Rod Johnson
- Who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002.
- The book was accompanied by 30,000 lines of framework code.
- Having already sacrificed almost a year's salary he put into writing the book. (Writing a 750 page book is enough work on its own).
- the framework named “Interface21” (at that point it used com.interface21 package names), but that was not a name to inspire a community.



Spring Releases Roadmap

- Shortly after the book was published, readers began to use the Wrox forums to discuss the code and two of them “Juergen Hoeller” and "Yann Caroffa” Persuaded Rod to make the code the basis of an open source project, and became co-founders.

- Juergen's name is of course central to any discussion of Spring today; but the Spring community should also remember Yann for his early contribution toward making the Spring project happen.

Spring Releases Roadmap

- Fortunately Yann stepped up with a suggestion: "Spring".
  - His reasoning was association with nature (having noticed that I'd trekked to Everest Base Camp in 2000); and the fact that Spring represented a fresh start after the “winter” of traditional J2EE.
  - They recognized the simplicity and elegance of this name, and quickly agreed on it.
- Yann eventually stopped contributing to open source to concentrate on playing music as a hobby and having a normal social life.
- The framework was first released under the Apache 2.0 license in June 2003.
- The Spring 1.2.6 framework won a Jolt productivity award and a JAX (Java API for XML) Innovation Award in 2006.

Spring Releases Roadmap

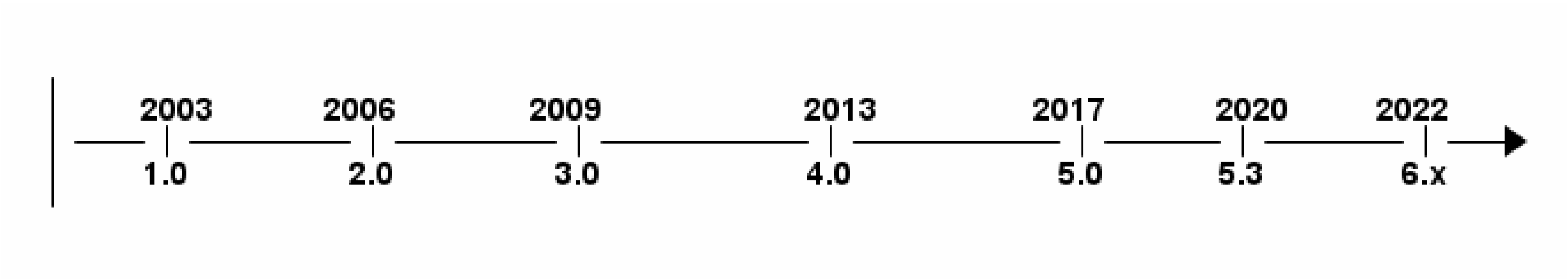
- Started in 2003 as a response to the complexity of the early J2EE specifications.
- Notable improvements in Spring 4.0 included support for Java SE (Standard Edition) 8, Groovy 2, some aspects of Java EE 7, and WebSocket.
- Spring Framework 4.2.1, which was released on 01 Sept 2015 and It is compatible with Java 6, 7 and 8, with a focus on core refinements and modern web capabilities
- Spring Framework 4.3 has been released on 10 June 2016 and will be supported until 2020.
  - It will be the final generation within the general Spring 4 system requirements (Java 6+, Servlet 2.5+)
- Spring 5 is announced to be built upon Reactive Streams compatible Reactor Core.

Spring Releases Roadmap

- It integrates with carefully selected individual specifications from the EE umbrella:
    - Dependency Injection (JSR 330)
    - Common Annotations (JSR 250)
    - Servlet API (JSR 340)
    - WebSocket API (JSR 356)
    - Concurrency Utilities (JSR 236)
    - JSON Binding API (JSR 367)
    - Bean Validation (JSR 303)
    - JPA (JSR 338)
    - JMS (JSR 914)

Spring Releases Roadmap

- Latest Spring release 6.x
  - (released on November 2022)



Feedback and Contributions

- For Questions or Diagnosing or Debugging issues:
  - They suggest using StackOverflow
  - Also they have question page that lists the suggested tags to use (https://spring.io/questions).
- If you’re certain that there is a problem in the Spring Framework or would like to suggest a feature they offer JIRA Issue Tracker on https://jira.spring.io/browse/spr
- If you want to contribute in Spring Framework see the guidelines at the CONTRIBUTING.

Jira Software

Lesson 3
Spring Framework introduction

What is Spring Framework?

- Spring is an open source Dependency Injection (DI) and Aspect Oriented Programming (AOP) lightweight container and full stack java EE application framework.

What is Spring Framework?

- Spring is an open source Dependency Injection (DI) and Aspect Oriented Programming (AOP) lightweight container and full stack java EE application framework.
- Dependency Injection (DI):
  - Don’t call us we’ll call you.
  - Service Injection rather than Service Lookup.
  - Reduces coupling between classes.
  - Supports Testing.

What is Spring Framework?

- Spring is an open source Dependency Injection (DI) and Aspect Oriented Programming (AOP) lightweight container and full stack java EE application framework.
- Aspect Oriented Programming (AOP):
  - enables organized development by separating business logic from system services (such as auditing or logging
  - allowing you to define method-interceptors and point cuts to decouple code implementing functionality.

What is Spring Framework?

- Spring is an open source Dependency Injection (DI) and Aspect Oriented Programming (AOP) lightweight container and full stack java EE application framework.
- lightweight:
  - Spring is lightweight in terms of size and overhead.
    - Doesn’t required special container to run spring Application.
    - Spring Framework Core modules is in jars their size about (weighs= 2 MB).
    - The processing overhead required by Spring is nothing.
  - Business objects in a Spring-enabled application often have no dependencies on Spring-specific classes.

What is Spring Framework?

- Spring is an open source Dependency Injection (DI) and Aspect Oriented Programming (AOP) lightweight container and full stack java EE application framework.
- Spring is a container in the sense:
  - It contains & manages the lifecycle & configuration of application objects.
  - You can declare how
    - Each objects should be created,
    - Objects should be configured,
    - Objects should be associated with each other.

Introduction Demos Outline

1) HelloWorld
2) HelloWorld with command line arguments
3) HelloWorld with decoupling using Classes
4) HelloWorld with decoupling using Interfaces
5) HelloWorld with decoupling through Factory
6) HelloWorld with Spring Framework & Dependency Injection (Properties)
7) HelloWorld with Spring Framework & Dependency Injection (XML)
8) HelloWorld with Spring Framework & Dependency Injection (Annotation)

Demo (1) HelloWorld

```java
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- <u>Problems:</u>
  - This code is not extensible.
  - You have to change code (and recompile) if you want to change the message.

Demo (2) HelloWorld with command line arguments

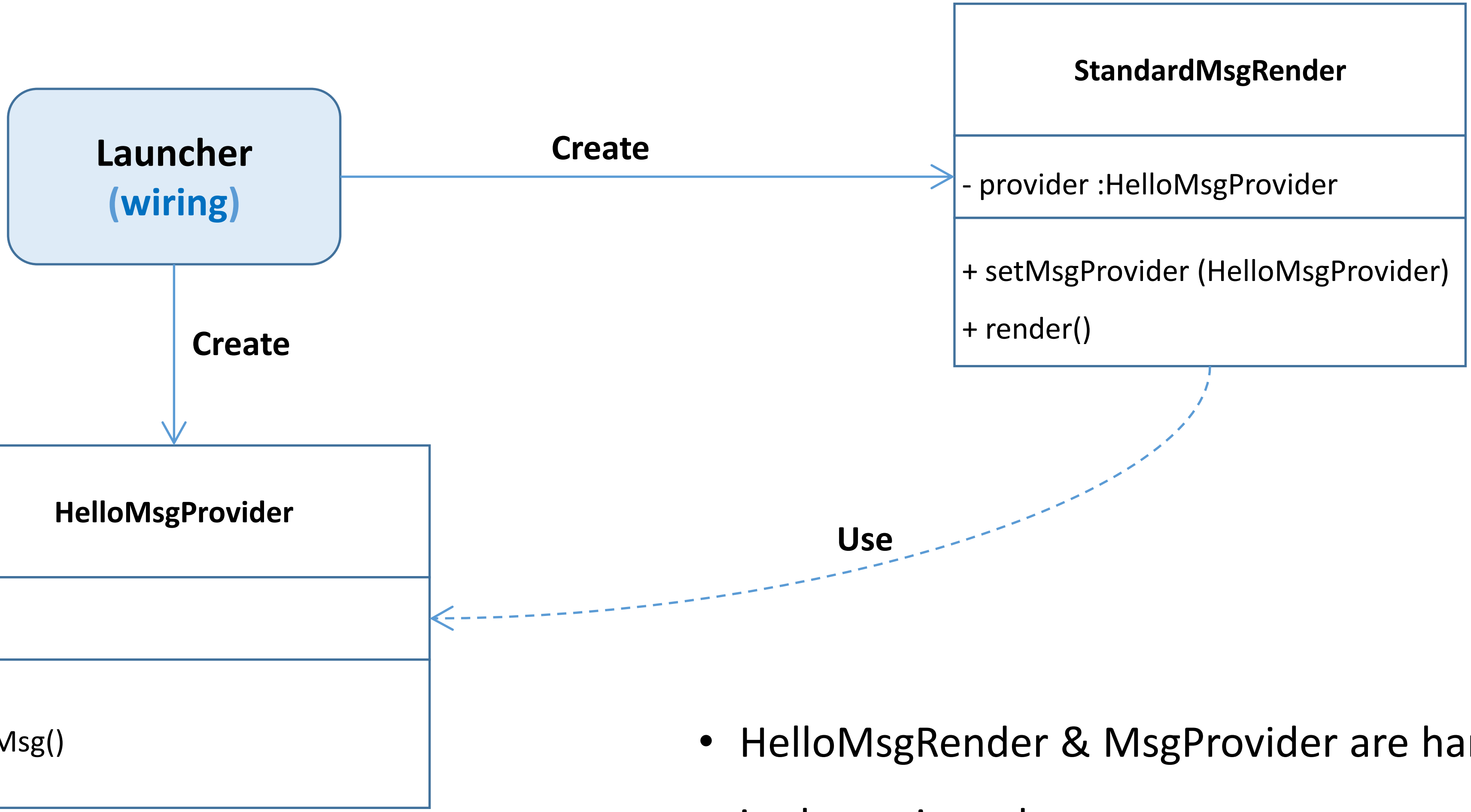
You can change the message without changing the code by passing arguments

```
public class HelloWorld {

    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Hello " + args[0]);
        } else {
            System.out.println("Hello World");
        }
    }
}
```

Demo (2) HelloWorld with command line arguments (Ex.)

- Problems:
  - The code responsible for:
    - The rendering message.
    - Obtaining the message.
- What if I want to output the message differently, may be viewing text in HTML tags rather than as plain text?

Demo (3) HelloWorld with decoupling using Classes



- HelloMsgRender & MsgProvider are hardcoded in the main code

Demo (3) HelloWorld with decoupling using Classes (Ex.)

- **<u>Message provider logic:</u>**
  - responsible for providing the message.
- **<u>Message rendering logic:</u>**
  - responsible for rendering the message.
- **<u>Launcher:</u>**
  - the main class that use message provider logic & message rendering logic.

Demo (3) HelloWorld with decoupling using Classes (Ex.)

```java
public class HelloMsgProvider {

    public String getMsg() {
        return "Hello World";
    }
}
```

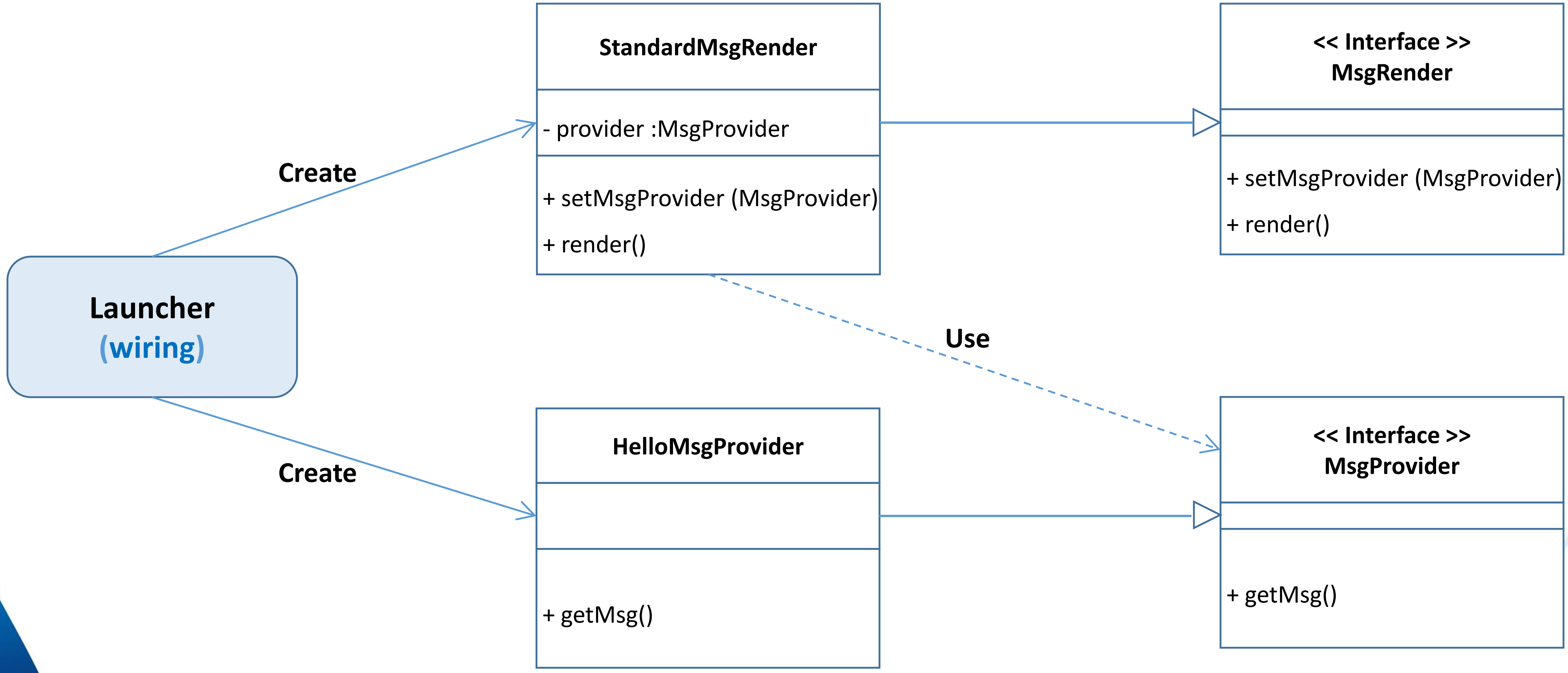
```java
public class StandardMsgRender {

    private HelloMsgProvider provider;

    public void setMsgProvider(HelloMsgProvider provider) {
        this.provider = provider;
    }

    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

```java
public class HelloWorld {

    public static void main(String[] args) {
        StandardMsgRender mr = new StandardMsgRender();
        HelloMsgProvider mp = new HelloMsgProvider();
        mr.setMsgProvider(mp);
        mr.render();
    }
}
```

Here We Create Instances from our services Classes

Here we Are wiring between the provider and renderer.

Demo (4) HelloWorld with decoupling using Interfaces



Demo (4) HelloWorld with decoupling using Interfaces (Ex.)

- <u>**Message provider logic:**</u>
  - Define the methods in an interface and the class that implements this interface.
  - Separated from Message Renderer.
- <u>**Message rendering logic:**</u>
  - Define the methods in an interface and the class that implements this interface.
  - Separated from Message Provider.
- <u>**Launcher:**</u>
  - The main class that use message provider logic & message rendering logic.
  - Which create instance from Message Renderer & Provider and wiring between them.

Demo (4) HelloWorld with decoupling using Interfaces (Ex.)

```java
public interface MsgProvider {

    public String getMsg();
}
```

```java
public class HelloMsgProvider
        implements MsgProvider {

    @Override
    public String getMsg() {
        return "Hello World";
    }
}
```

Demo (4) HelloWorld with decoupling using Interfaces (Ex.)

```
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```
public class StandardMsgRender
        implements MsgRender {
    private MsgProvider provider;
    @Override
    public void setMsgProvider(MsgProvider provider) {
        this.provider = provider;
    }
    @Override
    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

Demo (4) HelloWorld with decoupling using Interfaces (Ex.)

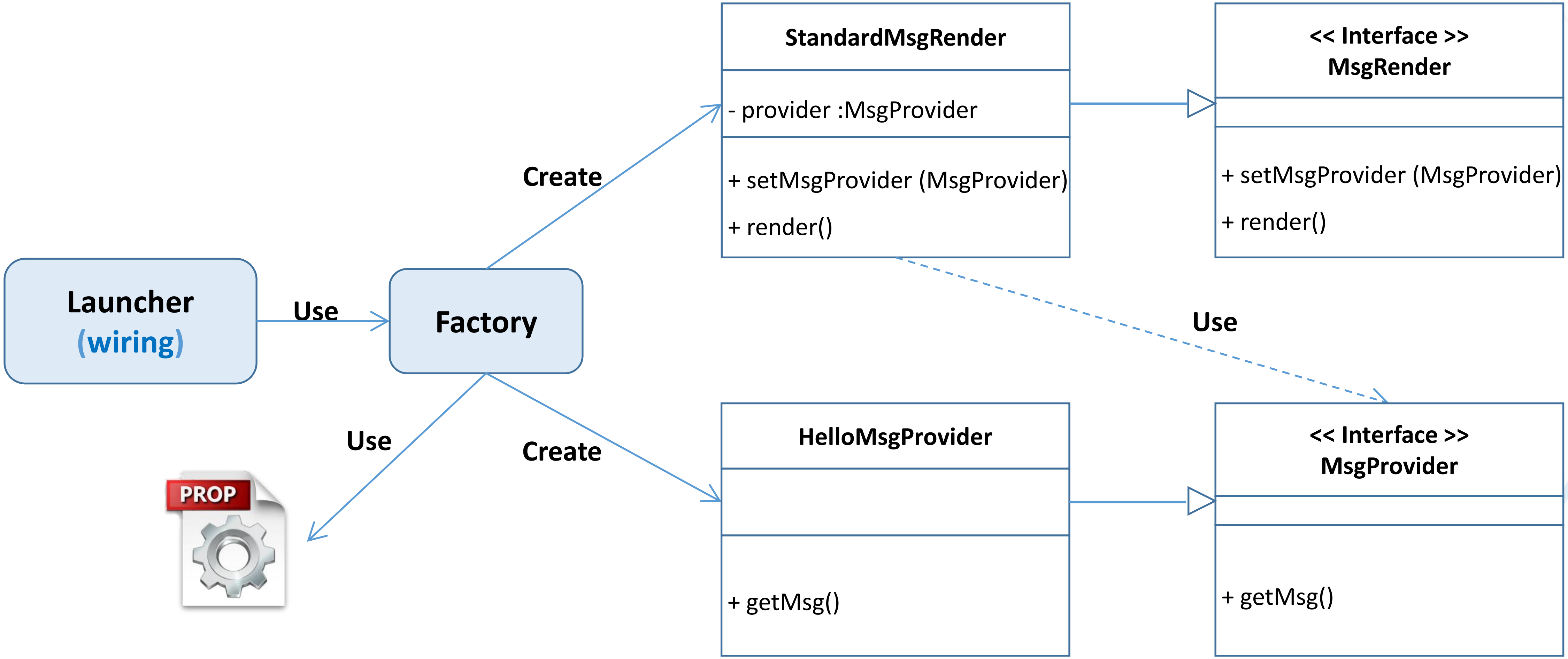
```
public class HelloWorld {

    public static void main(String[] args) {
        MsgRender mr = new StandardMsgRender();
        MsgProvider mp = new HelloMsgProvider();
        mr.setMsgProvider(mp);
        mr.render();
    }
}
```

Here We Create Instances from our services Classes

Here we Are wiring between the provider and renderer.

- Problems:
  - Using different implementation of *MsgRender interface* or *MsgProvider interface* means a change to the business logic code in launcher.

Demo (5) HelloWorld with decoupling through Factory



Demo (5) HelloWorld with decoupling through Factory (Ex.)

- **<u>Message provider logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Renderer.
- **<u>Message rendering logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Provider.
- **<u>Message Factory:</u>**
  - Which create instance from Message Renderer & Provider and wiring between them.
- **<u>Property File:</u>**
  - Has qualified name of Message Renderer and Message Provider.
- **<u>Launcher:</u>**
  - the main class that use message factory to get instances and wiring them.

Demo (5) HelloWorld with decoupling through Factory (Ex.)

```java
public interface MsgProvider {

    public String getMsg();
}
```

```java
public class HelloMsgProvider
        implements MsgProvider {

    @Override
    public String getMsg() {
        return "Hello World";
    }
}
```

Demo (5) HelloWorld with decoupling through Factory (Ex.)

```java
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```java
public class StandardMsgRender
        implements MsgRender {
    private MsgProvider provider;
    @Override
    public void setMsgProvider(MsgProvider provider) {
        this.provider = provider;
    }
    @Override
    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

Demo (5) HelloWorld with decoupling through Factory (Ex.)

```java
public class MessageSupportFactory {

    private static MessageSupportFactory instance = null;
    private Properties props = null;
    private MsgRender renderer = null;
    private MsgProvider provider = null;

    private MessageSupportFactory() {
        props = new Properties();
        try {
            props.load(MessageSupportFactory.class.getResourceAsStream("/msf.properties"));
            String renderClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");
            renderer = (MsgRender) Class.forName(renderClass).newInstance();
            provider = (MsgProvider) Class.forName(providerClass).newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

Demo (5) HelloWorld with decoupling through Factory (Ex.)

```java
    static {
        instance = new MessageSupportFactory();
    }

    public static MessageSupportFactory getInstance() {
        return instance;
    }

    public MsgRender getMsgRender() {
        return renderer;
    }

    public MsgProvider getMsgProvider() {
        return provider;
    }

}
```

```properties
# msf.properties
renderer.class=com.jediver.spring.introduction.demo.impl.StandardMsgRender
provider.class=com.jediver.spring.introduction.demo.impl.HelloMsgProvider
```

Demo (5) HelloWorld with decoupling through Factory (Ex.)

```
public class HelloWorld {

    public static void main(String[] args) {
        MessageSupportFactory factory =
                MessageSupportFactory.getInstance();
        MsgRender mr = factory.getMsgRender();
        MsgProvider mp = factory.getMsgProvider();
        mr.setMsgProvider(mp);
        mr.render();
    }
}
```

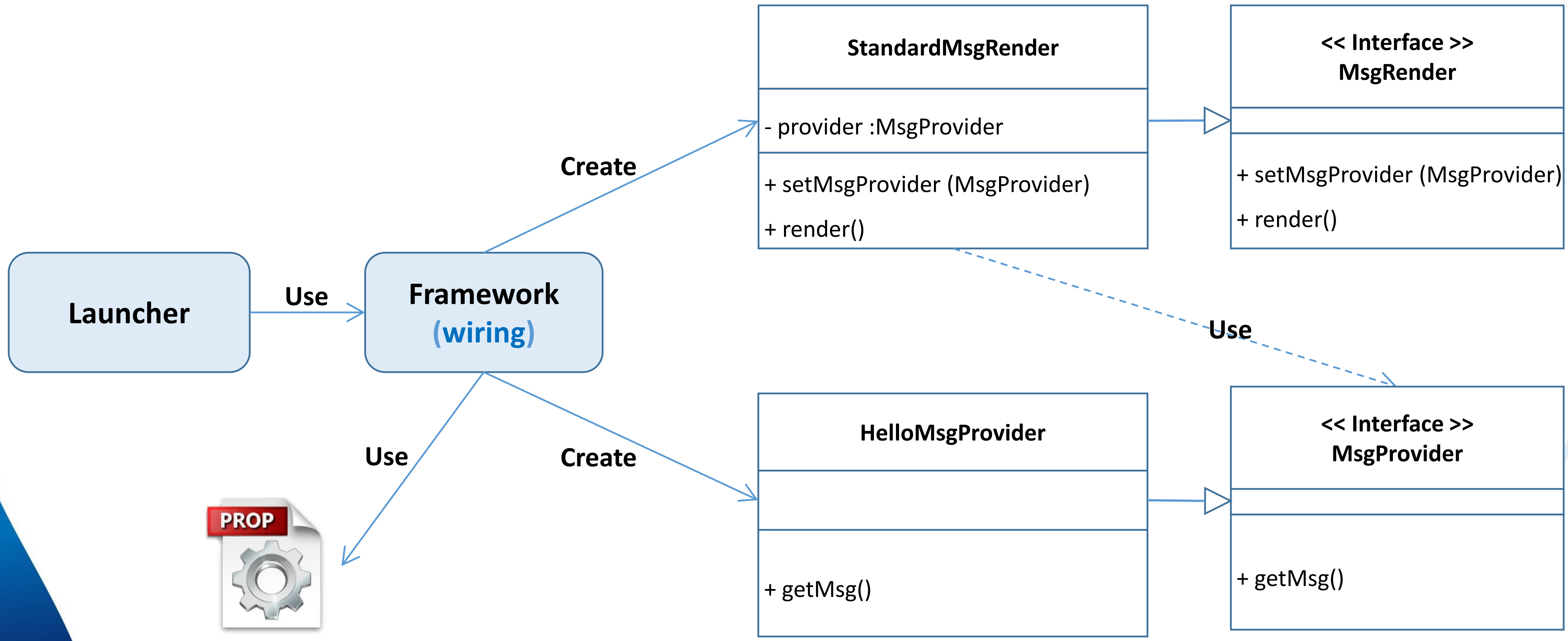
Create Instances from our singleton factory

Request Instances from our Factory

wiring between the provider and renderer.

- Problems:
  - You have to write a lot of glue code yourself (MessageSupportFactory) to pieces the application together.
  - You have to wire between MsgRender with an instance of MsgProvider manually.

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties)



Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

- **<u>Message provider logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Renderer.
- **<u>Message rendering logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Provider.
- **<u>Property File:</u>**
  - Has qualified name of Message Renderer and Message Provider.
- **<u>Launcher:</u>**
  - the main class that use message factory to get instances and wiring them.

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

```java
public interface MsgProvider {

    public String getMsg();
}
```

```java
public class HelloMsgProvider
        implements MsgProvider {

    @Override
    public String getMsg() {
        return "Hello World";
    }
}
```

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

```java
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```java
public class StandardMsgRender
        implements MsgRender {
    private MsgProvider provider;
    @Override
    public void setMsgProvider(MsgProvider provider) {
        this.provider = provider;
    }
    @Override
    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

```
# msf.properties
renderer.class=com.jediver.spring.introduction.demo.impl.StandardMsgRender
provider.class=com.jediver.spring.introduction.demo.impl.HelloMsgProvider
renderer.msgProvider(ref)=provider
```

Here We wire the created Instance of provider and renderer by calling method setMsgProvider() in renderer class and send provider as reference.

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

```java
    private static BeanFactory getBeanFactory() {
        // get the bean factory
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        // create a definition reader
        PropertiesBeanDefinitionReader rdr = new PropertiesBeanDefinitionReader(
                factory);
        // load the configuration options
        Properties props = new Properties();
        try {
            props.load(HelloWorld.class.getResourceAsStream("/msf.properties"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        rdr.registerBeanDefinitions(props);
        return factory;
    }
}
```

Demo (6) HelloWorld with Spring Framework & Dependency Injection (Properties) (Ex.)

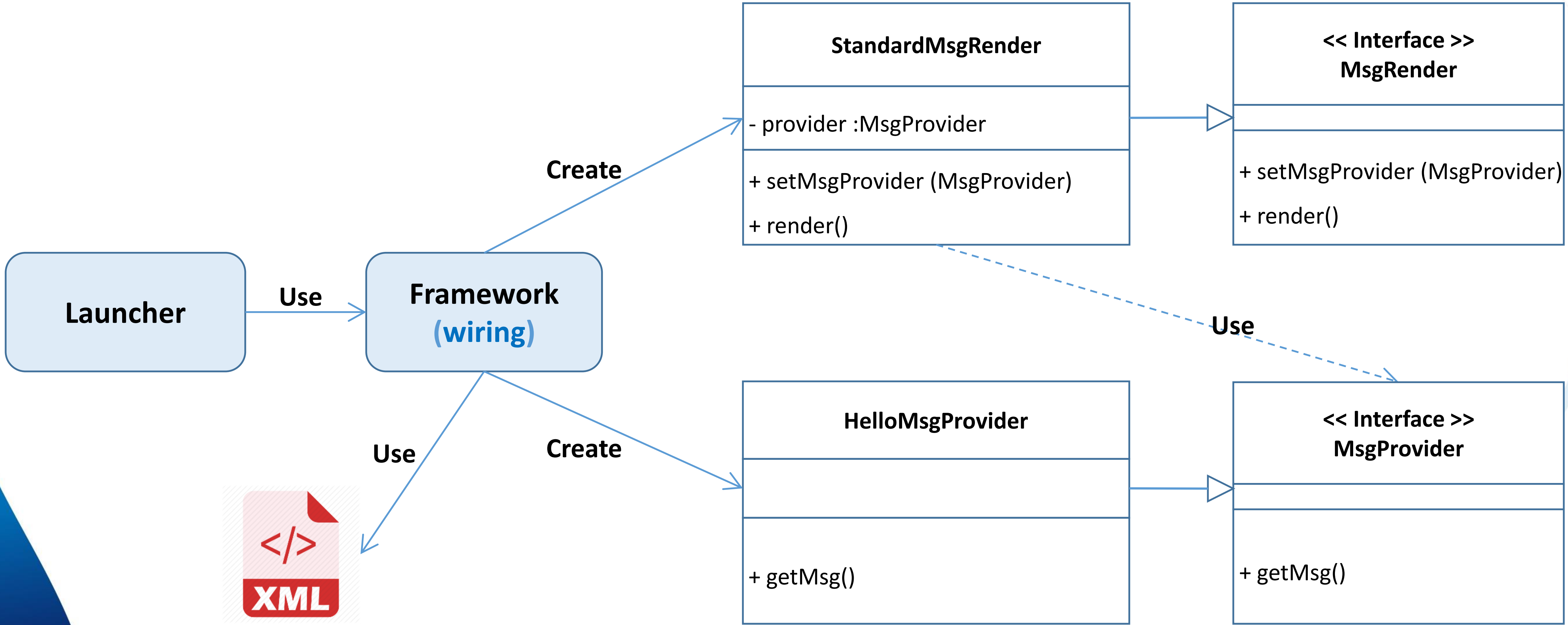
```java
public class HelloWorld {

    public static void main(String[] args) {
        // get the bean factory
        BeanFactory factory = getBeanFactory();
        MsgRender mr = (MsgRender) factory.getBean("renderer");
        // Note that you don't have to manually inject message provider to
        // message renderer anymore.
        mr.render();
    }
```

Get bean factory from local method

Request an instance from bean named renderer wired with message provider

- Problems:
  - Syntax to generate instances and wiring them in properties file is special and have many difficulties.

Demo (7)   HelloWorld with Spring Framework & Dependency Injection (XML)



Demo (7) HelloWorld with Spring Framework & Dependency Injection (XML) (Ex.)

- **<u>Message provider logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Renderer.
- **<u>Message rendering logic:</u>**
  - define the methods in an interface and the class that implements this interface, separated from Provider.
- **<u>XML File:</u>**
  - Has bean definitions and wiring syntax in structured way.
- **<u>Launcher:</u>**
  - the main class that use message factory to get instances and wiring them.

Demo (7) HelloWorld with Spring Framework & Dependency Injection (XML) (Ex.)

```java
public interface MsgProvider {

    public String getMsg();
}
```

```java
public class HelloMsgProvider
        implements MsgProvider {

    @Override
    public String getMsg() {
        return "Hello World";
    }
}
```

Demo (7) HelloWorld with Spring Framework & Dependency Injection (XML) (Ex.)

```java
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```java
public class StandardMsgRender
        implements MsgRender {
    private MsgProvider provider;
    @Override
    public void setMsgProvider(MsgProvider provider) {
        this.provider = provider;
    }
    @Override
    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

Demo (7) HelloWorld with Spring Framework & Dependency Injection (XML) (Ex.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="renderer"
          class="com.jediver.spring.introduction.demo.impl.StandardMsgRender">
        <property name="msgProvider">
            <ref bean="provider"/>
        </property>
    </bean>
    <bean id="provider"
          class="com.jediver.spring.introduction.demo.impl.HelloMsgProvider"/>
</beans>
```

Create instance of renderer

Wire between them by calling setMsgProvider()

Create instance of provider

Demo (7) HelloWorld with Spring Framework & Dependency Injection (XML) (Ex.)

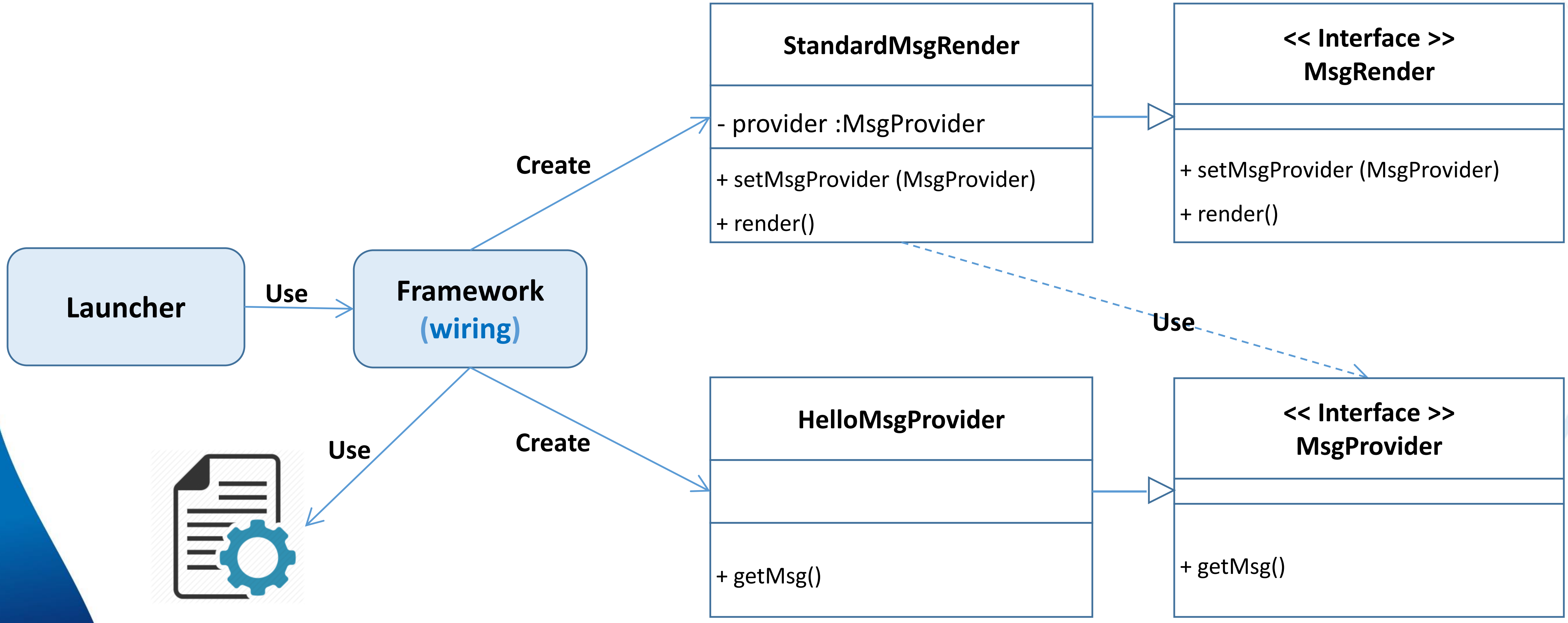
```java
public class HelloWorld {

    public static void main(String[] args) {
        // get the bean factory
        BeanFactory factory = getBeanFactory();
        MsgRender mr = (MsgRender) factory.getBean("renderer");

        // Note that you don't have to manually inject message provider
        // to message renderer anymore.
        mr.render();
    }

    private static BeanFactory getBeanFactory() {
        // get the bean factory
        BeanFactory factory
                = new XmlBeanFactory(new ClassPathResource("beans.xml"));
        return factory;
    }

}
```

Get bean factory from local method

Request an instance from bean named renderer wired with message provider

Create bean factory

Demo (8)   HelloWorld with Spring Framework & Dependency Injection (Annotation)



Demo (8) HelloWorld with Spring Framework & Dependency Injection (Annotation) (Ex.)

- **Message provider logic:**
  - define the methods in an interface and the class that implements this interface, separated from Renderer.
- **Message rendering logic:**
  - define the methods in an interface and the class that implements this interface, separated from Provider.
- **Configuration class File:**
  - Has bean definitions.
- **Launcher:**
  - the main class that use message factory to get instances and wiring them.

Demo (8) HelloWorld with Spring Framework & Dependency Injection (Annotation) (Ex.)

```java
public interface MsgProvider {

    public String getMsg();
}
```

```java
public class HelloMsgProvider
        implements MsgProvider {

    @Override
    public String getMsg() {
        return "Hello World";
    }
}
```

Demo (8) HelloWorld with Spring Framework & Dependency Injection (Annotation) (Ex.)

```
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```
public class StandardMsgRender implements MsgRender {
    @Autowired
    private MsgProvider provider;
    @Override
    public void setMsgProvider(MsgProvider provider) {
        this.provider = provider;
    }
    @Override
    public void render() {
        System.out.println(provider.getMsg());
    }
}
```

Demo (8) HelloWorld with Spring Framework & Dependency Injection (Annotation) (Ex.)

Create instance of provider

Create instance of renderer

```java
@Configuration
public class HelloWorldConfig {
    @Bean
    public MsgProvider createMsgProvider() {
        return new HelloMsgProvider();
    }
    @Bean
    public MsgRender createMsgRender() {
        return new StandardMsgRender();
    }

}
```

Demo (8) HelloWorld with Spring Framework & Dependency Injection (Annotation) (Ex.)

```
public class HelloWorld {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx =
                new AnnotationConfigApplicationContext();

        ctx.register(HelloWorldConfig.class);
        ctx.refresh();

        MsgRender mr = ctx.getBean(MsgRender.class);
        mr.render();
    }
}
```

Create Empty Context that can understand annotation

Register Configuration Class & refresh the context

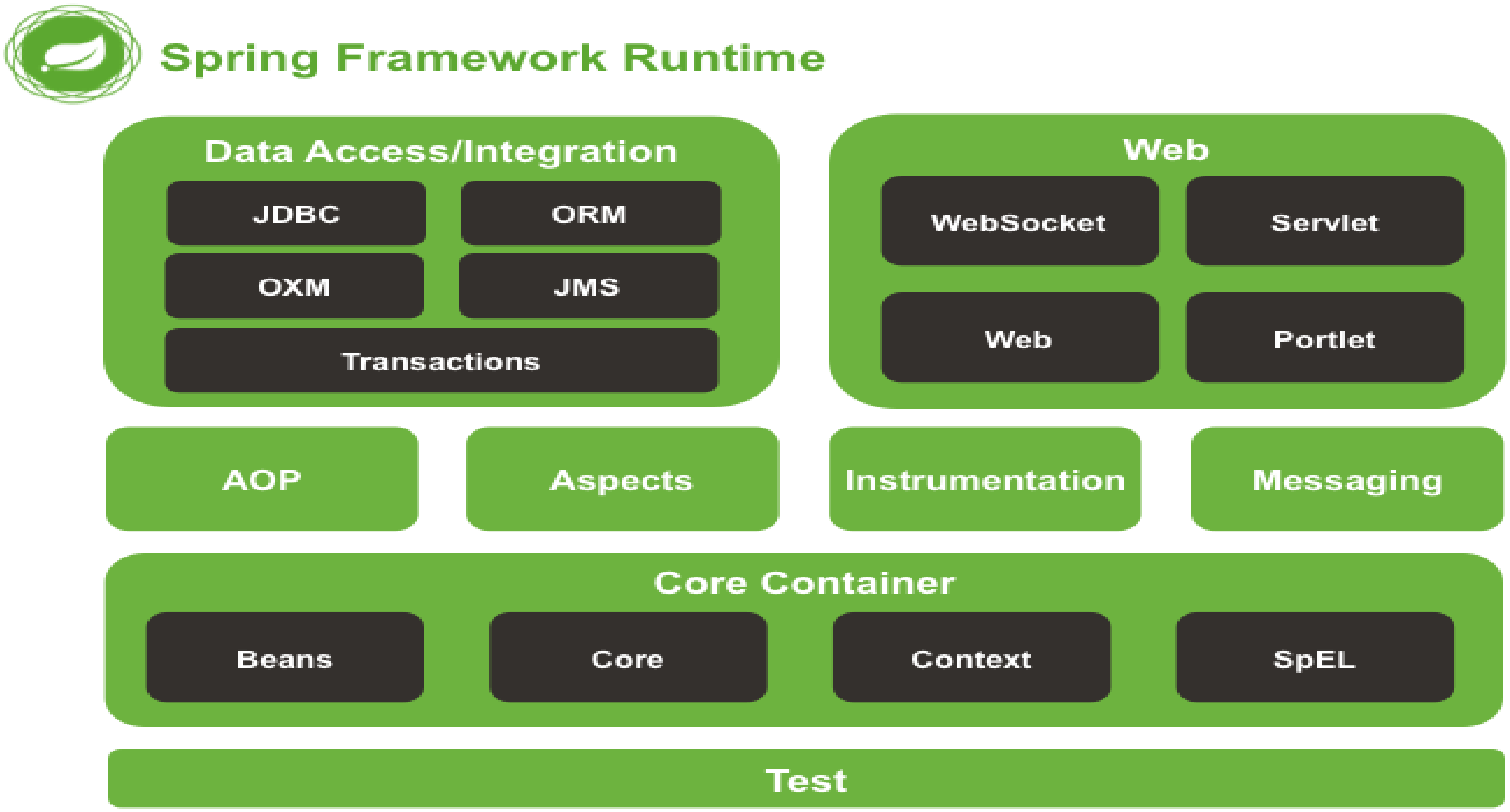
Request an instance from bean named renderer wired with message provider

Lesson 4

Spring 6.x Modules

(released on November 2022)

Spring Framework Modules



Lesson 5

Core Container (Bean Overview)

The IoC container

- Stands for Inversion of Control
- IoC is also known as *dependency injection* (DI).
- It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean.
- Spring Framework's IoC container base packages:

org.springframework.beans and **org.springframework.context**

The IoC container (Ex.)

<< Interface >>

BeanFactory

- **org.springframework.beans.factory**
- provides an advanced configuration mechanism capable of managing any type of object.

<< Interface >>
ApplicationContext

- **is a sub-interface of BeanFactory**
- integration with Spring’s AOP features
- Supports Annotation
- message resource handling (for use in internationalization
- event publication
- application-layer specific contexts such as the WebApplicationContext for use in web applications
- Integration with **any other module**.

The IoC container (Ex.)

- **Spring IoC container:**
  - Its has 3 responsibilities:
    - Instantiating
    - Assembling
    - Managing
  - The most used BeanFactory implementation is the XmlBeanFactory class.
    - This implementation expresses the objects in terms of XML.
    - Become deprecated in Latest versions we can use create by BeanDefinitionRegistry.
- BeanFactory Interface can do only the three responsibilities of IoC Container.
- The **XmlBeanFactory** takes XML configuration metadata only and uses it to create a fully configured system or application.

Demo BeanFactory

- **Calculator interface:**

```
public interface Calculator {

    public double add(double num1, double num2);

    public double subtract(double num1, double num2);

    public double multiply(double num1, double num2);

    public double divide(double num1, double num2);
}
```

Demo BeanFactory (Ex.)

- **Calculator Implementation:**

```java
public class CalculatorImpl implements Calculator {

    @Override
    public double add(double num1, double num2) {
        return num1 + num2;
    }

    @Override
    public double subtract(double num1, double num2) {
        return num1 - num2;
    }
    @Override
    public double multiply(double num1, double num2) {
        return num1 * num2;
    }

    @Override
    public double divide(double num1, double num2) {
        if (num2 == 0) {
            throw new RuntimeException("Invalid Second operand cannot equals zero.");
        }
        return num1 / num2;
    }
}
```

Demo BeanFactory (Ex.)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="calculatorID"
            class="com.jediver.spring.core.calculator.impl.CalculatorImpl"/>
</beans>
```

Create instance of CalculatorImpl with id “calculatorID”

Demo BeanFactory (Ex.)

```java
public class Application {

    public static void main(String[] args) {
        // get the bean factory
        BeanFactory factory
                = new XmlBeanFactory(new ClassPathResource("beans.xml"));
        Calculator calculator = (Calculator) factory.getBean("calculatorID");
        // Note that you don't have to manually inject message provider
        // to message renderer anymore.
        System.out.println(calculator.add(2, 3));
        System.out.println(calculator.subtract(2, 3));
        System.out.println(calculator.multiply(2, 3));
        System.out.println(calculator.divide(2, 0));
    }
}
```

Create bean factory

Request an instance from bean named calculatorID

Call Calculator Methods

The IoC container (Ex.)

- **We can use the un-deprecated way by:**

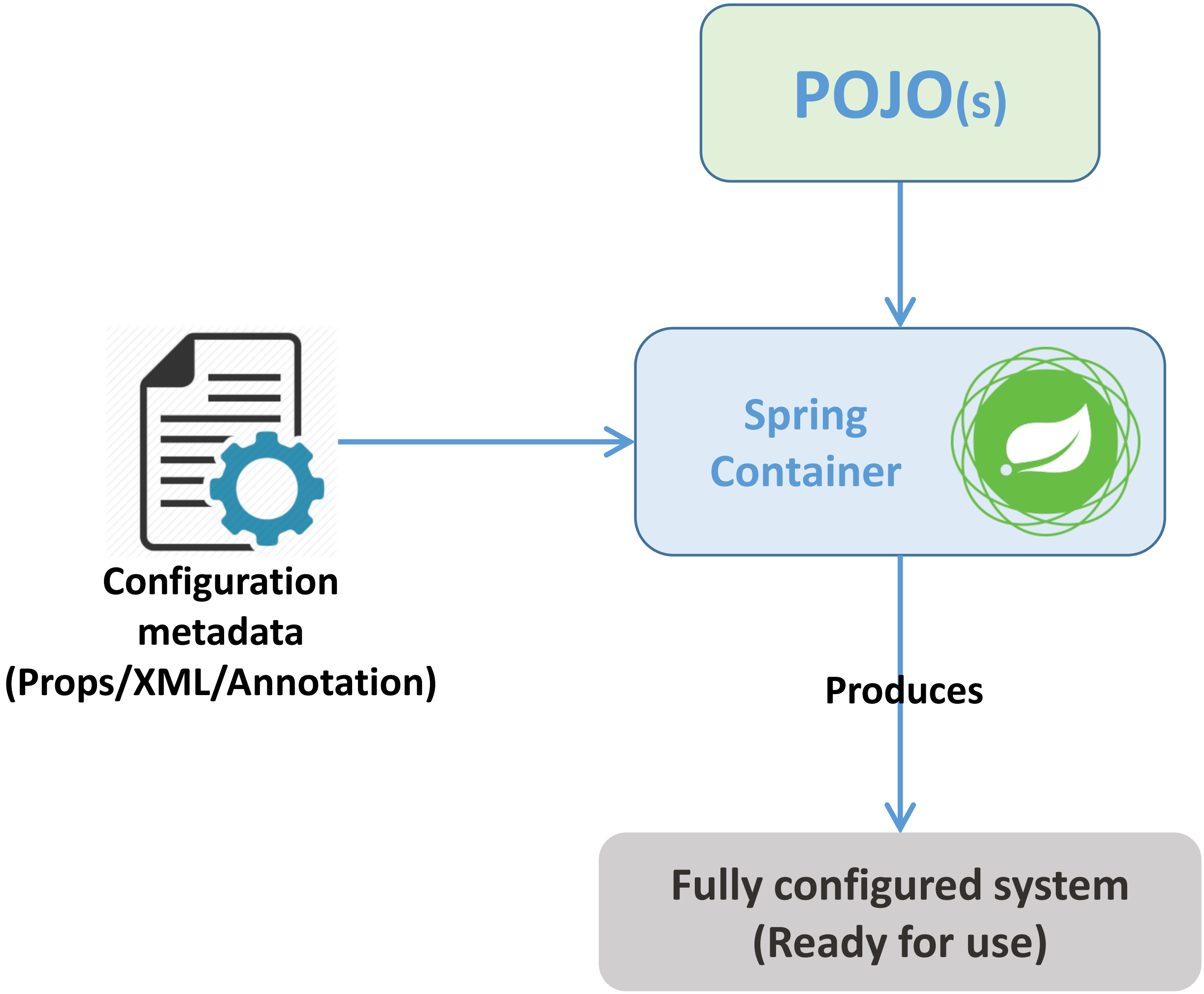
```java
BeanDefinitionRegistry beanDefinitionRegistry
        = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader
        = new XmlBeanDefinitionReader(beanDefinitionRegistry);
reader.loadBeanDefinitions(new ClassPathResource("beans.xml"));
BeanFactory factory = (BeanFactory) beanDefinitionRegistry;
```

The IoC container (Ex.)

- **We can use Application context interface by:**

```
ApplicationContext factory
        = new ClassPathXmlApplicationContext("beans.xml");
Calculator calculator = (Calculator) factory.getBean("calculatorID");
```

The IoC container (Ex.)



Composing XML-based File(s)

- It is useful to split up container definitions into multiple XML files so it can be reused in another application.
- There are two approach to compose the XML file:
  - Use the application context constructor which takes multiple Resource locations

```
ApplicationContext context =
        new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

Composing XML-based File(s)

- Use one or more occurrences of the <import/> element to load bean definitions from another file(s).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <import resource="services.xml"/>
    <import resource="daos.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>
</beans>
```

Bean Definition

- Managed Beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions).
- Each bean definition consists of set of.
  - Attributes:
    - id
    - name
    - class
    - parent
    - scope
    - abstract
    - lazy-init
    - autowire
    - depends-on
    - autowire-candidate
    - primary
    - init-method
    - destroy-method
    - factory-method
    - factory-bean

Bean Definition (Ex.)

- Each bean definition consists of set of.
  - Sub-elements
    - constructor-arg
    - description
    - lookup-method
    - meta
    - property
    - qualifier
    - replaced-method

Bean Definition (Ex.)

- The 'id' attribute:
  - the value of id attribute must be unique per document.
- The 'name' attribute:
  - You may specify one or more bean name, separated by
  - **Comma (,)** and/or
  - **Semicolon (;)** and/or
  - **Whitespace**.
- The extra names of bean can be considered aliases.

Bean Definition (Ex.)

- Aliasing a Bean outside the Bean Definition
- Why we need Alias?
  - It is useful for some situations, such as allowing each component to refer to a common dependency using a bean name that is specific to that component itself.
- use <alias/> element.
  - `<alias name="fromName" alias="toName" />`

Instantiating Beans

- The 'class' attribute:
  - Specify the class of the bean to be constructed where the container itself directly creates the bean by:
    1. **Instantiating using Constructor.**
    2. **Instantiating using static factory method.**
    3. **Instantiating using factory method in factory bean.**
  - This 'class' attribute is normally mandatory

Instantiating Beans (Ex.)

1. Instantiating using Constructor.

- Calling default constructor (without <constructor-arg/>)

```xml
<bean id="service1"
      class="com.jediver.spring.core.bean.defination.impl.Service1Impl"/>
```

- Calling constructor that takes int argument

```xml
<bean id="service2"
      class="com.jediver.spring.core.bean.defination.impl.Service1Impl">
    <constructor-arg value="20"/>
</bean>
```

- Calling constructor that takes string argument

```xml
<bean id="service3"
      class="com.jediver.spring.core.bean.defination.impl.Service1Impl">
    <constructor-arg value="Medhat"/>
</bean>
```

Instantiating Beans (Ex.)

2. Instantiating using static factory method.

- The class containing the static factory method that is to be invoked to create the object by attribute named ‘factory-method’ is needed to specify the name of the factory method itself.
- Calling factory method that didn’t take any parameters (without <constructor-arg/>)

```
<bean id="service5"
        class="com.jediver.spring.core.bean.defination.impl.Service1Impl"
        factory-method="createService1Impl">
</bean>
```

- Calling factory method that takes int argument

```
<bean id="service6"
        class="com.jediver.spring.core.bean.defination.impl.Service1Impl"
        factory-method="createService1Impl">
    <constructor-arg value="20"/>
</bean>
```

Instantiating Beans (Ex.)

3. Instantiating using factory method in factory bean.

- mean a non-static method of an existing bean from the container is invoked to create a new bean.
- The 'class' attribute must be **left empty**,
- The 'factory-bean' attribute must specify the name of a bean in the container that contains the instance method
- Factory instance must be created by spring container itself.

```
<bean id="factory"
        class="com.jediver.spring.core.bean.defination.impl.ServiceFactory">
</bean>
```

Instantiating Beans (Ex.)

3. Instantiating using factory method in factory bean.

- Calling factory method that didn't take any parameters (without <constructor-arg/>)

```xml
<bean id="service7"
      factory-bean="factory"
      factory-method="createService1Impl">
</bean>
```

- Calling factory method that takes int argument

```xml
<bean id="service8"
      factory-bean="factory"
      factory-method="createService1Impl">
    <constructor-arg value="20"/>
</bean>
```

Lesson 6

Core Container (Dependancies)

Dependency Injection

- A process whereby objects define their dependencies (that is, the other objects with which they work) only through:
  1. **Constructor Based Injection**
  2. **Factory Method Based Injection**
  3. **Setter Based Injection**
- Then, the container inject those dependencies when it creates the bean

Dependency Injection (Ex.)

1. Constructor Based Injection

- Invoke a constructor with a number of arguments.
- Constructor argument resolution matching occurs using the argument's type.
- Calling Constructor that take one parameter

```java
public class DAOServiceImpl implements DAOService {

    private ProductDAO productDAO;
    private UserDAO userDAO;

    public DAOServiceImpl(ProductDAO productDAO) {
        this.productDAO = productDAO;
    }
}
```

Dependency Injection (Ex.)

1. Constructor Based Injection

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="productDAO"
          class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
    <bean id="daoService"
          class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <constructor-arg ref="productDAO"/>
    </bean>
</beans>
```

Create standalone bean

Refer to the created bean in constructor

OR

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="daoService"
          class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <constructor-arg>
            <bean
                class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
        </constructor-arg>
    </bean>
</beans>
```

Create anonymous bean (without id or name) and inject it to constructor

Dependency Injection (Ex.)

1. Constructor Based Injection

- Calling Constructor that take two parameter.

```
public class DAOServiceImpl implements DAOService {

    private ProductDAO productDAO;
    private UserDAO userDAO;

    public DAOServiceImpl(UserDAO userDAO, ProductDAO productDAO) {
        this.productDAO = productDAO;
        this.userDAO = userDAO;
    }
}
```

Dependency Injection (Ex.)

1. Constructor Based Injection

- Either you can define bean outside/inside the new bean.

```
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="userDAO"
          class="com.jediver.spring.core.bean.di.impl.UserDAOImpl"/>
    <bean id="daoService"
          class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <constructor-arg ref="userDAO"/>
        <constructor-arg>
            <bean
                class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
        </constructor-arg>
    </bean>
</beans>
```

Create standalone bean

Refer to the created bean in constructor

Create anonymous bean (without id or name) and inject it to constructor

Dependency Injection (Ex.)

1. Constructor Based Injection (Constructor Argument Index)

- Constructor arguments can have their index specified explicitly by use of the index attribute.
- Specifying an index solves the problem of ambiguity where a constructor may have two arguments of the same type.
- **Note:** the index is 0 Based.

```java
public class User {

    private String name;
    public User(String firstName, String lastName) {
        this.name = firstName + lastName;
    }
}
```

```xml
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
```

Dependency Injection (Ex.)

1. Constructor Based Injection (Constructor Argument Type)

- When a simple type is used, Spring cannot determine the type of the value, and so cannot match by type without help.
- So you can use type matching with simple types by using the 'type' attribute

```java
public class User {

    private String name;
    private float balance;

    public User(String name, float balance) {
        this.name = name;
        this.balance = balance;
    }
}
```

```xml
<bean id="user"
      class="com.jediver.spring.core.bean.di.User">
    <constructor-arg type="float" value="1222"/>
    <constructor-arg type="java.lang.String" value="Medhat"/>
</bean>
```

Dependency Injection (Ex.)

2. Factory Method Based Injection

- Invoke a Factory method with a number of arguments.
- Factory method argument resolution matching occurs using the argument's type.
- Calling a Factory method that take one parameter

```java
public class ServiceFactory {

    public DAOService createDAOService(ProductDAO productDAO) {
        return new DAOServiceImpl(productDAO);
    }
}
```

Dependency Injection (Ex.)

2. Factory Method Based Injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="factory"
            class="com.jediver.spring.core.bean.di.impl.ServiceFactory">
    </bean>
    <bean id="productDAO"
            class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
    <bean id="daoService"
            factory-bean="factory"
            factory-method="createDAOService">
        <constructor-arg ref="productDAO"/>
    </bean>
</beans>
```

Create standalone bean

Refer to the created bean in constructor

Dependency Injection (Ex.)

2. Factory Method Based Injection

- Calling Constructor that take two parameter.

```
public class ServiceFactory {

    public DAOService createDAOService(UserDAO userDAO,
            ProductDAO productDAO) {
        return new DAOServiceImpl(userDAO, productDAO);
    }

}
```

Dependency Injection (Ex.)

2. Factory Method Based Injection

- Either you can define bean outside/inside the new bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="factory"
        class="com.jediver.spring.core.bean.di.impl.ServiceFactory">
    </bean>
    <bean id="userDAO"
        class="com.jediver.spring.core.bean.di.impl.UserDAOImpl"/>
    <bean id="daoService"
        factory-bean="factory"
        factory-method="createDAOService">
        <constructor-arg ref="userDAO"/>
        <constructor-arg>
            <bean
            class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
        </constructor-arg>
    </bean>
</beans>
```

Create standalone bean

Create anonymous bean (without id or name) and inject it to constructor

Refer to the created bean in constructor

Dependency Injection (Ex.)

3. Setter Based Injection

- is realized by calling setter methods on your beans after construct bean either with empty or parameterized Constructor
- After we construct the object we call the setter of object "productDAO".

```java
public class DAOServiceImpl implements DAOService {

    private ProductDAO productDAO;

    public void setProductDAO(ProductDAO productDAO) {
        this.productDAO = productDAO;
    }
}
```

Dependency Injection (Ex.)

3. Setter Based Injection

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="productDAORef"
          class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
    <bean id="daoService"
          class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <property name="productDAO" ref="productDAORef"/>
    </bean>
</beans>
```

Create standalone bean

Refer to the created bean in property

OR

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="daoService"
          class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <property name="productDAO" >
            <bean
                class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
        </property>
    </bean>
</beans>
```

Create anonymous bean (without id or name) and inject it to property setter

Dependency Injection (Ex.)

3. Setter Based Injection

- After we construct the object we call the setter of object "productDAO" & "userDAO" .

```java
public class DAOServiceImpl implements DAOService {

    private ProductDAO productDAO;
    private UserDAO userDAO;

    public void setProductDAO(ProductDAO productDAO) {
        this.productDAO = productDAO;
    }

    public void setUserDAO(UserDAO userDAO) {
        this.userDAO = userDAO;
    }
}
```

Dependency Injection (Ex.)

3. Setter Based Injection

- Either you can define bean outside/inside the new bean.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean id="userDAORef"
            class="com.jediver.spring.core.bean.di.impl.UserDAOImpl"/>
    <bean id="daoService"
            class="com.jediver.spring.core.bean.di.impl.DAOServiceImpl">
        <property name="userDAO" ref="userDAORef"/>
        <property name="productDAO" >
            <bean
                class="com.jediver.spring.core.bean.di.impl.ProductDAOImpl"/>
        </property>
    </bean>
</beans>
```

Create standalone bean

Refer to the created bean in property setter

Create anonymous bean (without id or name) and inject it to property setter

Constructor-based or setter-based DI?

- We can mix constructor-based and setter-based DI.
- It is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.
  - **Note:** we can use of the @Required annotation on a setter method can be used to make the property be a required dependency.
- The **Spring team** generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not null.
  - **Side Note:** A large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Constructor-based or setter-based DI?

- Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class.
- One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later.
- Use the DI style that makes the most sense for a particular class.
- Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you.
  - For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

Circular dependencies

- If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.
  - For example:
    - **Class A** requires an instance of **Class B** through constructor injection.
    - And **Class B** requires an instance of **Class A** through constructor injection.
    - If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.
- One possible solution is to edit the source code of some classes to be configured by setters rather than constructors.
- Alternatively, avoid constructor injection and use setter injection only. In other words, although it is possible, it is not recommended.

Straight Values vs. p-namespace

- **Straight Values (Primitives, Strings, and so on)**
  - Spring's conversion service is used to convert property values from a String to the type of the property.

```xml
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <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="masterkaoli"/>
</bean>
```

OR Using p-namespace

```xml
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="com.mysql.jdbc.Driver"
      p:url="jdbc:mysql://localhost:3306/mydb"
      p:username="root"
      p:password="masterkaoli"/>
```

The idref element

- If you want to pass a bean id to another bean.

```xml
<bean id="adminUserId" class="com.jediver.spring.core.bean.di.User"/>
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <property name="name">
        <idref bean="adminUserId"/>
    </property>
</bean>
```

Any bean from any class

Injected the id Value not the bean itself

OR

```xml
<bean id="adminUserId" class="com.jediver.spring.core.bean.di.User"/>
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="adminUserId"/>
</bean>
```

- Using idref is preferable, because it lets the container validate at deployment time that the referenced, named bean actually exists.

References to Other Beans (Collaborators)

- The final element inside a <constructor-arg/> or <property/> definition element.
- You set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container.
- Scoping and validation of collaborators as follows:
- bean (most general form) `<ref bean="collaboratorRef"/>`
  - Any bean in the same container or parent container
  - Regardless of whether it is in the same XML file
  - Could be **bean id** or with this **bean name**
- local `<ref local="collaboratorRef"/>`
  - In the same XML file
  - is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular bean reference any more.

References to Other Beans (Collaborators)(EX.)

- parent
  - Any bean in a parent container
  - Regardless of whether it is in the same XML file
  - Could be **bean id** or with this **bean name**
  - Most common used in AOP (Context for beans without aspects and another with aspects)

```xml
<bean id="accountService" class="com.something.SimpleAccountService">
</bean>
<bean id="accountService"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>
    </property>
</bean>
```

Inner beans

- A <bean/> element inside the <property/> or <constructor-arg/> elements.
- An inner bean definition does not need to have any id or name defined.
  - **Note:** in case of inner beans,
    - The 'scope' flag is ignored.
    - Inner beans are always scoped as prototypes.
    - It is not possible to inject inner beans into beans other than the enclosing bean.

```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name">
        <bean class="java.lang.String">
            <constructor-arg value="Ahmed Medhat"/>
        </bean>
    </property>
</bean>
```

Collections

- The Collections are:
  - <list/> element.
  - <set/> element.
  - <map/> element.
  - <props/> element.

Collections <list/> element.

```xml
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails2">
        <list>
            <value>Hello World</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

Collections <set/> element.

```xml
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails4">
        <set>
            <value>Hello World</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

Collections <map/> element.

```
<bean id="parent"
        class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails3">
        <map>
            <entry key="administrator" value="administrator@example.com"/>
            <entry key="support" value="support@example.com"/>
        </map>
    </property>
</bean>
```

Collections <props/> element.

```xml
<bean id="parent"
        class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
```

Null Element

- Empty arguments for properties like as empty Strings.
- The following configuration sets the name property to the empty String value ("").

```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value=""/>
</bean>
```

- The <null/> element is used to handle null values.

```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" >
        <null/>
    </property>
</bean>
```

XML Shortcuts

- The <property/>, <constructor-arg/>, and <entry/> elements
- All support a 'value' attribute which may be used instead of embedding a full <value/> element.
- Examples:

```xml
<property name="name" >
    <value>Ahmed Medhat</value>
</property>
```

```xml
<property name="name"
    value="Ahmed Medhat"/>
```

```xml
<constructor-arg>
    <value>Ahmed Medhat</value>
</constructor-arg>
```

```xml
<constructor-arg
    value="Ahmed Medhat"/>
```

```xml
<entry key="key1">
    <value>Ahmed Medhat</value>
</entry>
```

```xml
<entry key="key1"
    value="Ahmed Medhat"/>
```

XML Shortcuts (Ex.)

- The <property/>, <constructor-arg/> support a 'ref' attribute which may be used instead of embedding a full <ref/> element.
- The <entry/> elements allows a shortcut form to specify the key and/or value of the map, in the form of the 'key' / 'key-ref' and 'value' / 'value-ref' attributes
- Examples:

```xml
<property name="name" >
    <ref bean="bean1"/>
</property>
```

⇓

```xml
<property name="name"
        ref="bean1"/>
```

```xml
<constructor-arg>
    <ref bean="bean1"/>
</constructor-arg>
```

⇓

```xml
<constructor-arg
    ref="bean1"/>
```

```xml
<entry key="key1">
    <key>
        <ref bean="keyBean"/>
    </key>
    <ref bean="bean1"/>
</entry>
```

⇓

```xml
<entry key-ref="keyBean"
    value-ref="bean1"/>
```

XML Shortcut with the p-namespace

- The p-namespace lets you use the bean element’s attributes (instead of nested <property/> elements) to describe your property values collaborating beans, or both.
- Import namespace (xmlns:p="http://www.springframework.org/schema/p")
- The following Example shows the standard XML format

```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="Ahmed Medhat"/>
    <property name="email" value="eng.medhat.cs.h@gmail.com"/>
</bean>
```

- The following Example shows the uses of p-namespace

```
<bean id="user" class="com.jediver.spring.core.bean.di.User"
      p.name="Ahmed Medhat"
      p.email="eng.medhat.cs.h@gmail.com"/>
```

XML Shortcut with the p-namespace (Ex.)

- This next example includes two more bean definitions that have a reference to another bean:
- The following Example shows the standard XML format

```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="Ahmed Medhat"/>
    <property name="email" value="eng.medhat.cs.h@gmail.com"/>
    <property name="department" ref="departmentBean"/>
</bean>
```

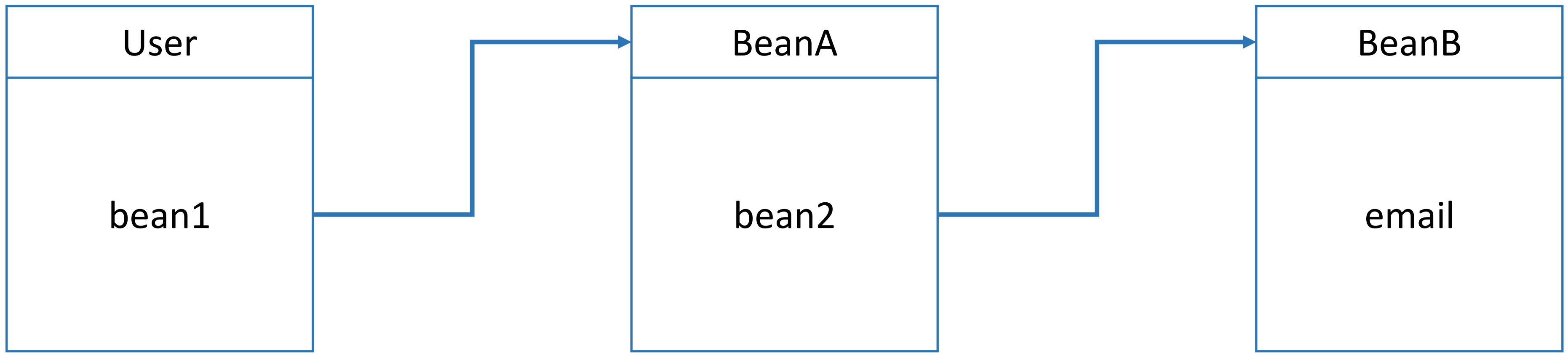
- The following Example shows the uses of p-namespace

```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User"
        p.name="Ahmed Medhat"
        p.email="eng.medhat.cs.h@gmail.com"
        p.department-ref="departmentBean"/>
```

- In this case, department is the property name, whereas the -ref part indicates that this is not a straight value but rather a reference to another bean.

Compound property names

- You can use compound or nested property names when you set bean properties.
- As long as all components of the path except the final property name are not null.
- The following Example:



Compound property names

- The following Example:

```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="bean1.bean2.email"
              value="eng.medhat.cs.h@gmail.com" />
</bean>
```

- The user bean has a bean1 property, which has a bean2 property, which has a email property.
- And that final email property is being set to a value of "eng.medhat.cs.h@gmail.com".
- In order for this to work, the bean1 property of user and the bean2 property of bean1 must not be null after the bean is constructed. Otherwise, a NullPointerException is thrown.

Using depends-on

- If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the <ref/> element in XML-based configuration metadata.
- However, sometimes dependencies between beans are less direct.
- An example is when a static initializer in a class needs to be triggered, such as for database driver registration.
- The "depends-on" attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.

Using depends-on (Ex.)

- The following example uses the depends-on attribute to express a dependency on a single bean:

```xml
<bean id="userDao" class="com.jediver.spring.dal.dao.UserDAO"
      depends-on="connection"/>
<bean id="connection" class="com.jediver.spring.dal.cfg.Connection"/>
```

- We can use (commas, whitespace, and semicolons) as delimiters to express a dependency on multiple beans, Another Example:

```xml
<bean id="bean1" class="com.jediver.spring.BeanOne"
      depends-on="connection,userDao"/>
<bean id="userDao" class="com.jediver.spring.dal.dao.UserDAO"/>
<bean id="connection" class="com.jediver.spring.dal.cfg.Connection"/>
```

Autowiring collaborators

- The Spring container can autowire relationships between collaborating beans.
- You can let Spring resolve collaborators (other beans) automatically for your bean by ApplicationContext.
- Autowiring has the following advantages:
    - Reduce the need to specify properties or constructor arguments.
    - Update a configuration as your objects evolve.
        - For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration.

Autowiring collaborators (Ex.)

- You can specify the autowire mode for a bean definition with the "autowire" attribute of the <bean/> element.
- The autowiring functionality has four modes:

  1. **no (default)**
     - (Default) No autowiring.
     - Bean references must be defined by ref elements.
     - Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity.

Autowiring collaborators (Ex.)

- The autowiring functionality has four modes:

  2. byName

  - Autowiring by property name.
  - Container Wiring a bean with the same name as the property that needs to be autowired.
  - For example, if a bean definition is set to autowire by name and it contains a userDao property (that is, it has a setUserDao(..) method), Spring looks for a bean definition named userDao and uses it to set the property.

Autowiring collaborators (Ex.)

- The autowiring functionality has four modes:

  3. byType

  - Autowiring by property type
  - Container Wiring if exactly one bean of the property type exists in the container.
  - If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean.
  - If there are no matching beans, nothing happens (the property is not set).

Autowiring collaborators (Ex.)

- The autowiring functionality has four modes:

  4. **constructor**

     - Autowiring by property type in constructor arguments
     - Container Wiring byType but applies to constructor arguments.
     - If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

Autowiring collaborators (Ex.)

- The following Example using autowiring "byName":
- UserService Class contains property called "userDao".

```
<bean id="userService"
      class="com.jediver.spring.service.UserService"
      autowire="byName"/>

<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"/>
```

- Container will look for bean named as "userDao" and wire this instance with the property "userDao" by calling userService.setUserDao(userDao);

Autowiring collaborators (Ex.)

- With byType or constructor autowiring mode, you can wire arrays and typed collections.
- In such cases, all autowire candidates within the container that match the expected type are provided to satisfy the dependency.
  - For Example: List<UserDao> userDaos;
  - Container will look for any bean with type UserDao and wire it into this list.
- You can autowire instances in Map if the expected key type is String.
  - For Example: Map<String, UserDao> userDaos;
  - Container will look for any bean with type UserDao and wire it into this map with name as key.
- An autowired instances based on:
  - Map instance's values consist of all bean instances that match the expected type.
  - And the Map instance's keys contain the corresponding bean names.

Autowiring collaborators (Ex.)

- **Advantages of Autowiring**
  - Autowiring can reduce the amount of configuration required:
  - You just write ‘autowire’ attribute only, and You don’t need to manually inject beans into each other by write <Property> or <constructor-arg> elements.
  - Autowiring can cause configuration to keep itself up to date as your objects evolve:
    - Fully Integrated with Observer Design Pattern.
    - Like add new definition for BeanA, it autowired in List<BeanA>

Autowiring collaborators (Ex.)

- **Limitations and Disadvantages of Autowiring**
  - Explicit dependencies in property and constructor-arg settings always override autowiring.
  - You cannot autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties).
  - Autowiring is less exact than explicit wiring. Spring is careful to avoid guessing in case of ambiguity that might have unexpected results.
    - The relationships between your Spring-managed objects are no longer documented explicitly.

Autowiring collaborators (Ex.)

- **Limitations and Disadvantages of Autowiring (Ex.)**
  - Wiring information may not be available to tools that may generate documentation from a Spring container.
  - Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired.
    - For arrays, collections, or Map instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

Autowiring collaborators (Ex.)

- **Resolve The Autowiring disadvantage**
  - Don’t use autowiring in favor of explicit wiring.
  - Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false.
  - Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
  - Implement the more fine-grained control available with annotation-based configuration.
  - Limit autowire candidates based on pattern-matching against bean names, by define top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute.

Manage a Bean in Autowiring

- You can exclude a bean from autowiring.
- By the autowire-candidate attribute of the <bean/> element to false.
- The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as @Autowired).
- The autowire-candidate attribute is designed to only affect type-based autowiring.
- It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate.

Manage a Bean in Autowiring (Ex.)

- You can manage bean in wiring by:

1. Excluding a bean from being an autowire candidate by

```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      autowire-candidate="false"/>
```

2. Making a bean the only candidate for autowiring by

```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      primary="true"/>
```

Manage a Bean in Autowiring (Ex.)

- You can manage bean in wiring by:

  3. Limit autowire candidates based on pattern-matching against bean names, by define top-level

     <beans/> element accepts one or more patterns within its default-autowire-candidates attribute

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-autowire-candidates="*Dao">
```

  - Limit the autowiring of beans in which its names ends with "Dao".

Manage a Bean in Autowiring (Ex.)

- You can mix one or more management.
- Limit the autowiring of beans in which its names ends with "Dao" or "Service" or "Util" .

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd"
        default-autowire-candidates="*Dao,*Service,*Util">
```

- Exclude this bean "userDao" from autowire candidates.

```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        autowire-candidate="false"/>
```

Lesson 7

Core Container (Bean Scopes)

Bean Scopes

- When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition.
- The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.
- You can control the scope of the objects created from a particular bean definition.
- It gives you the flexibility to choose the scope of the objects through configuration
- You can define scope by "scope" attribute in <bean />

```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      scope="prototype"/>
```

Bean Scopes

- The Spring Framework supports six scopes:
  - You can also create a custom scope.

1. **singleton**
   - (Default) Scopes a single bean definition to a single object instance for each Spring container.
2. **prototype**
   - Scopes a single bean definition to any number of object instances.
3. **request**
   - Scopes a single bean definition to the lifecycle of a single HTTP request.
   - That is, each HTTP request has its own instance of a bean created off the back of a single bean definition.
   - Only valid in the context of a web-aware Spring ApplicationContext.

Bean Scopes

- The Spring Framework supports six scopes:

4. **session**
   - Scopes a single bean definition to the lifecycle of an HTTP Session.
   - Only valid in the context of a web-aware Spring ApplicationContext.
5. **application**
   - Scopes a single bean definition to the lifecycle of a ServletContext.
   - Only valid in the context of a web-aware Spring ApplicationContext.
6. **websocket**
   - Scopes a single bean definition to the lifecycle of a WebSocket.
   - Only valid in the context of a web-aware Spring ApplicationContext.

Bean Scopes

1. **singleton**

    - The singleton scope is the default scope in Spring.
    - Only one shared instance of a singleton bean is managed.
    - When you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition.
    - This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
    - A singleton from Spring Container VS. A singleton as defined in the Gang of Four (GoF) patterns.
        - The GoF singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader.
        - The scope of the Spring singleton is best described as being per-container and per-bean.

Bean Scopes

1. **singleton**

- The singleton scope is the default scope in Spring.

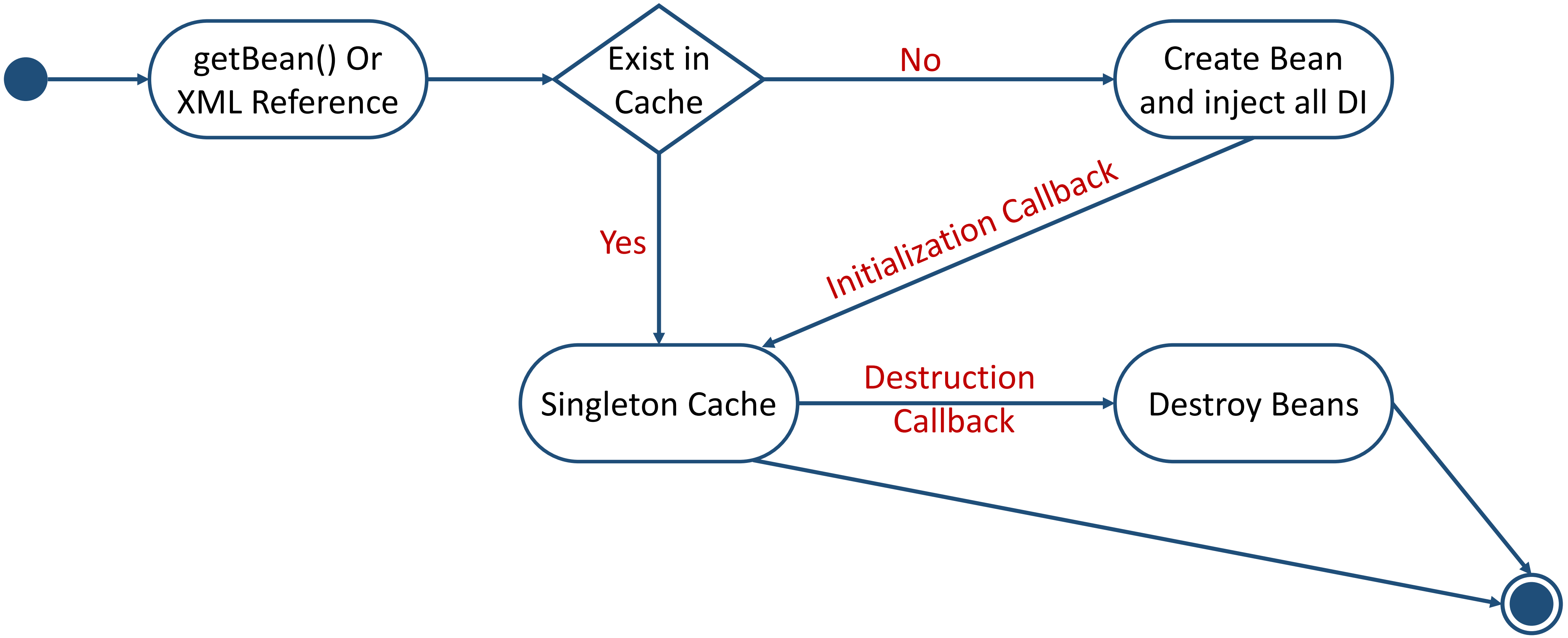
```xml
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"/>
```

- To define a bean as a singleton in XML, you can define a bean as shown in the following example:

```xml
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      scope="singleton"/>
```

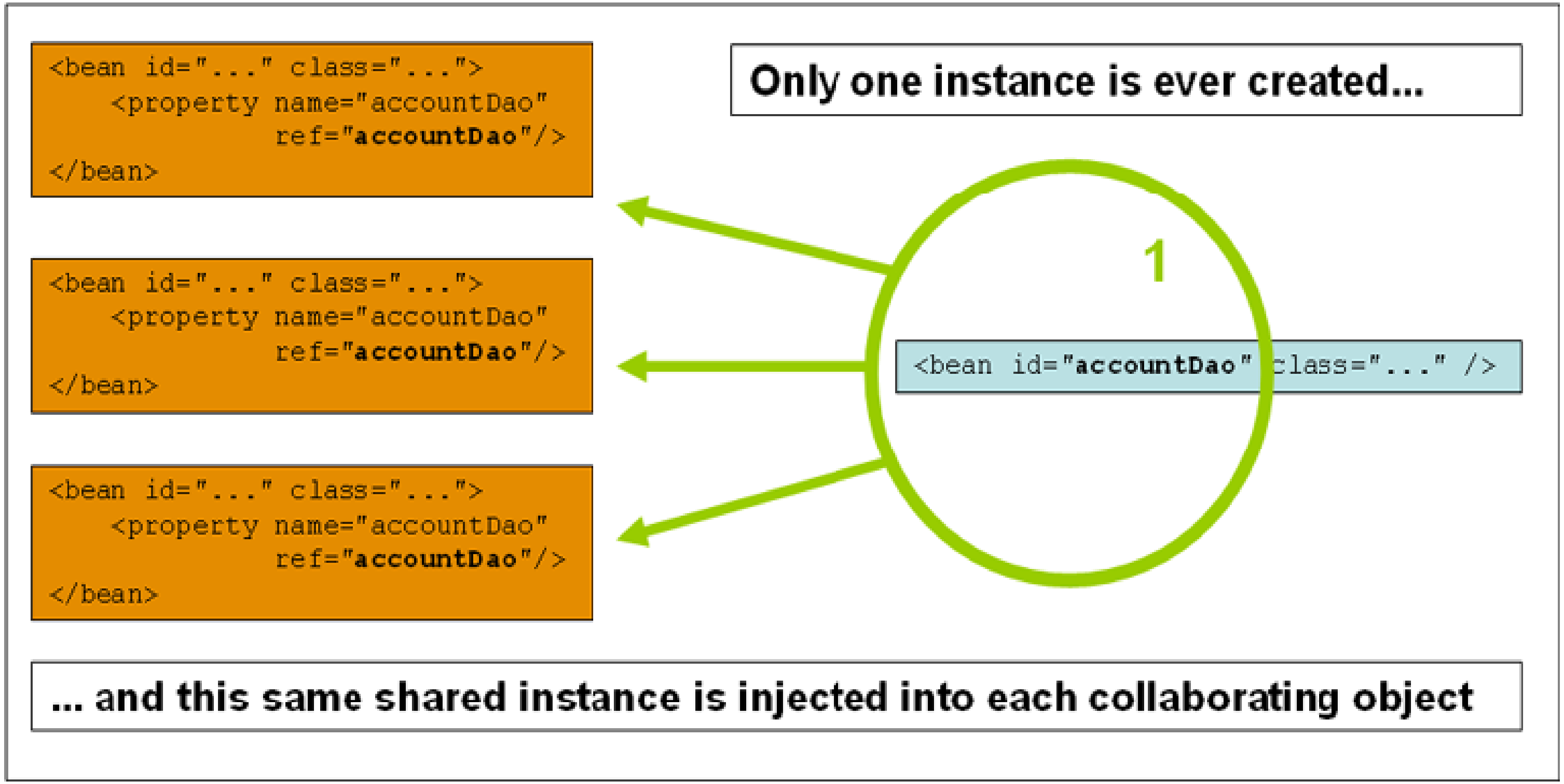
Bean Scopes

1. **singleton (Lifecycle)**



Bean Scopes

1. **singleton**



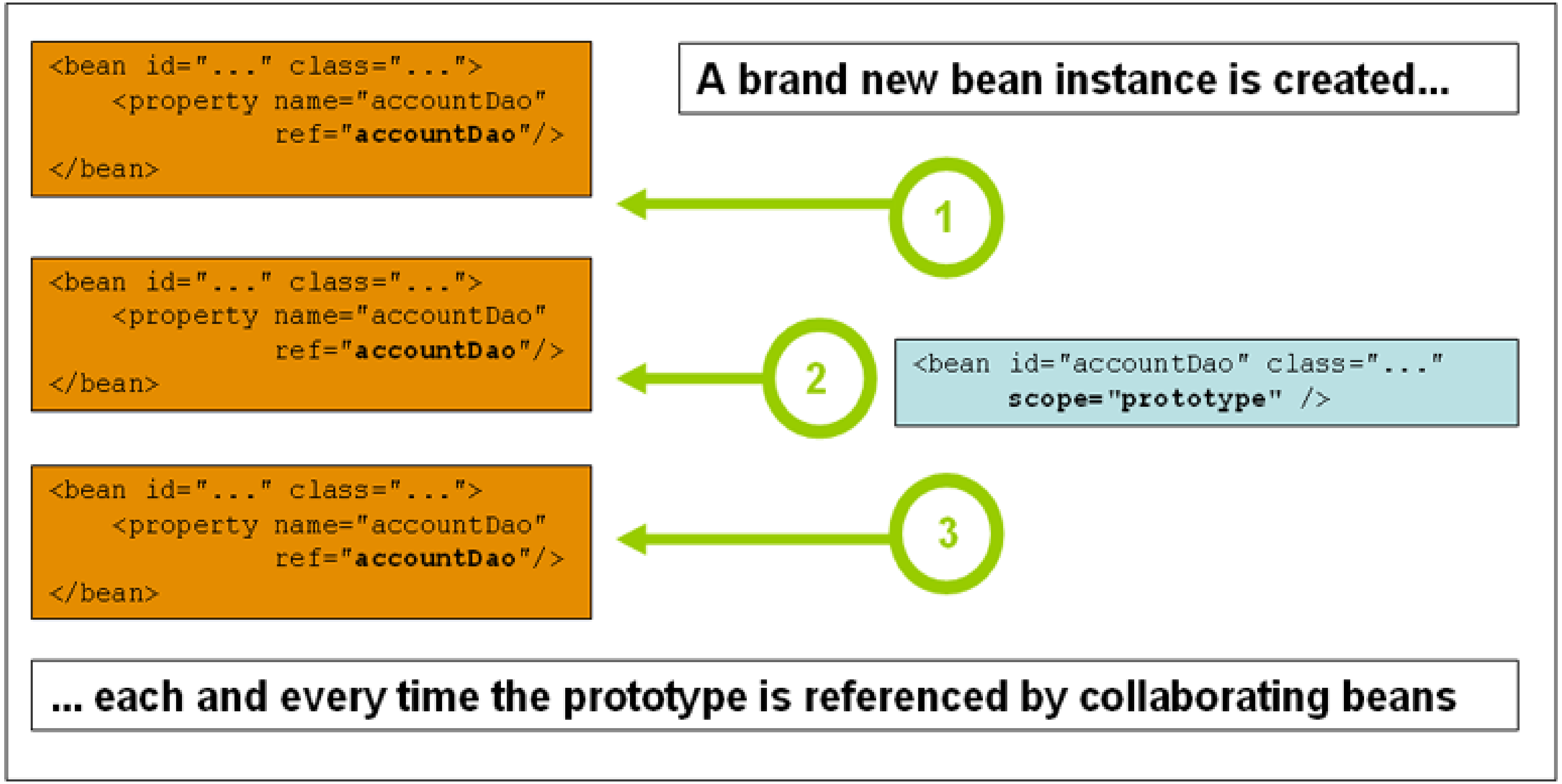
Bean Scopes

2. **prototype**

    - The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made.
    - That is, the bean is injected into another bean or you request it through a getBean() method call on the container.
    - As a rule:
        - You should use the prototype scope for stateful beans.
        - You Should use the singleton scope for stateless beans.

Bean Scopes

2. prototype



Bean Scopes

2. prototype

- The following example defines a bean as a prototype in XML:

```
<bean id="userService"
      class="com.jediver.spring.service.UserService"
      scope="prototype" />
```

- For Example:
  - A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state.

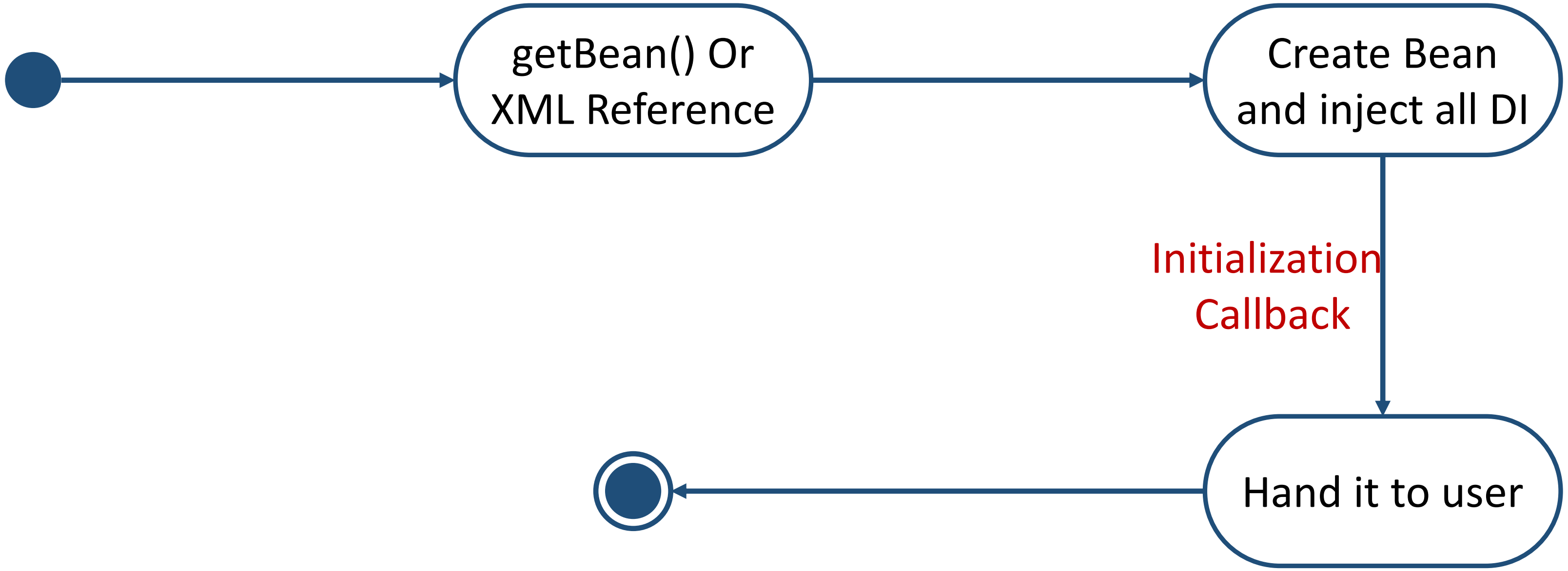
Bean Scopes

2. **prototype**

- In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean.
- The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance.
- Initialization lifecycle callback methods are called on all objects regardless of scope.
- In the case of prototypes, Configured destruction lifecycle callbacks are not called.
- The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold.
- To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

Bean Scopes

2. **prototype (Lifecycle)**



Bean Scopes

- **Singleton Beans with Prototype-bean Dependencies**
  - When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time.
  - If you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean.
  - However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime.
    - You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container instantiates the singleton bean and resolves and injects its dependencies.
    - If you need a new instance of a prototype bean at runtime more than once, use **Method Injection**.

Bean Scopes

- The **request**, **session**, **application**, and **websocket** scopes are available only if you use a web-aware Spring ApplicationContext implementation.
- If you use these scopes with regular Spring IoC containers, such as the ClassPathXmlApplicationContext, an IllegalStateException that complains about an unknown bean scope is thrown.

Bean Scopes

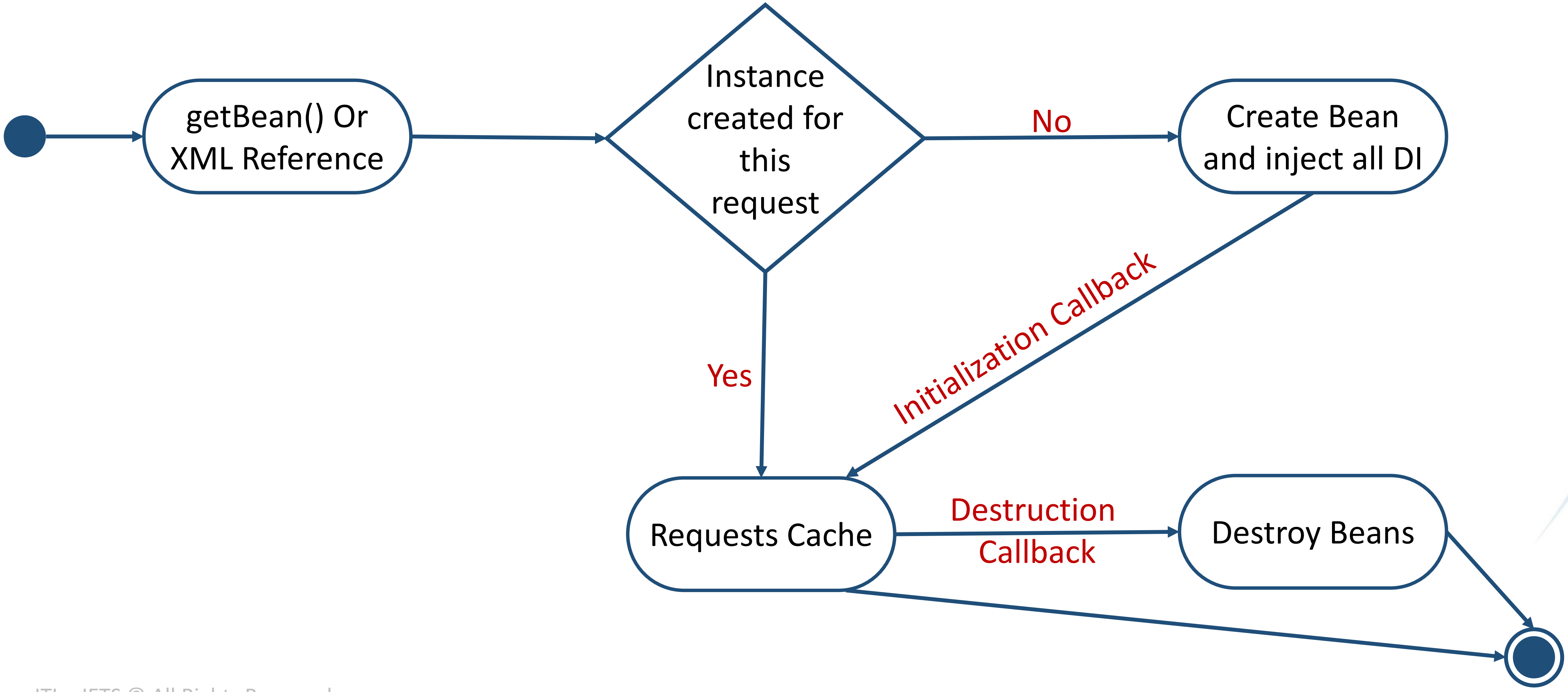
3. request

- The Spring container creates a new instance of the UserController bean by using the userController bean definition for each and every HTTP request **(Stateful Per Request)**.
- You can change the internal state of the instance that is created as much as you want, because other instances created from the same userController bean definition do not see these changes in state.
- They are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.
- Consider the following XML configuration for a bean definition:

```
<bean id="userController"
      class="com.jediver.spring.view.controller.UserController"
      scope="request" />
```

Bean Scopes

3. request (Lifecycle)



Bean Scopes

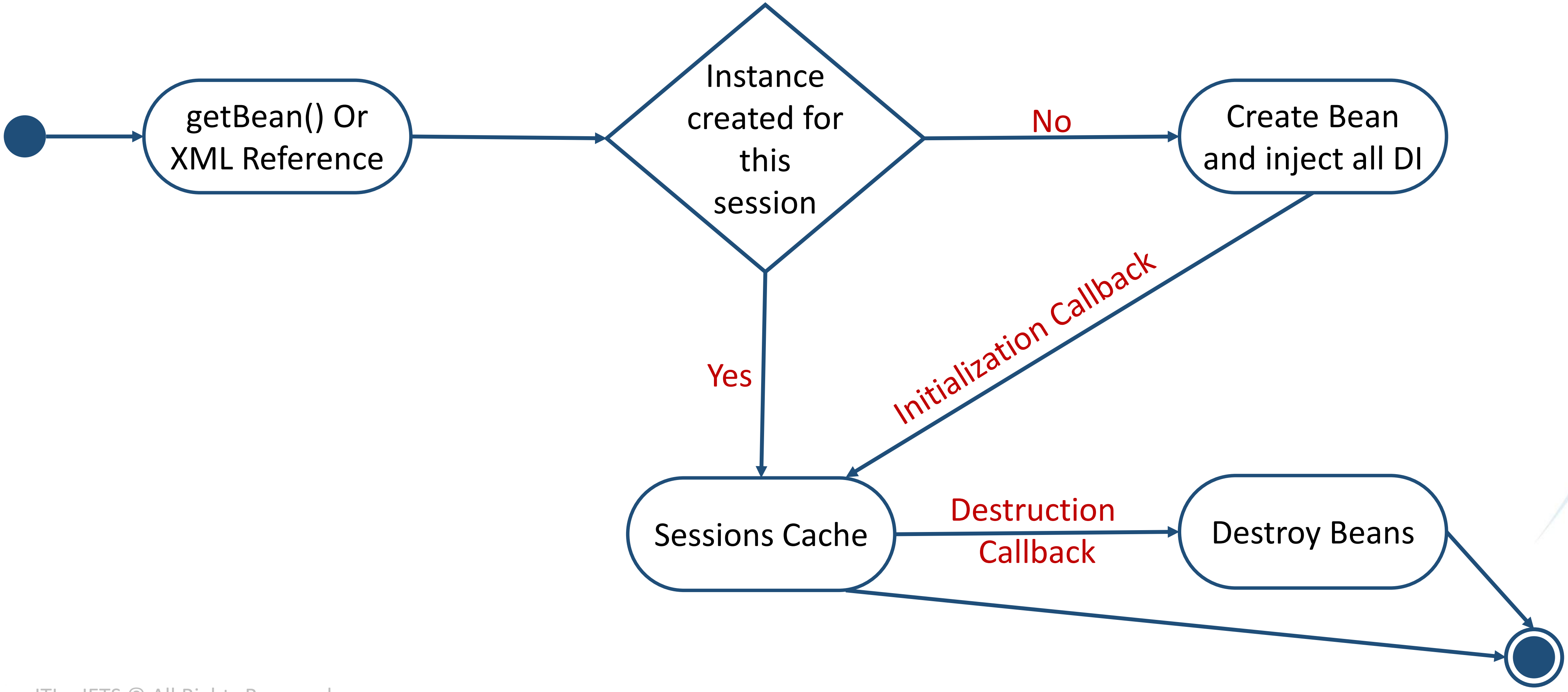
4. **session**

- The Spring container creates a new instance of the UserCart bean by using the userCart bean definition for the lifetime of a single HTTP Session **(Stateful per Session)**.
- When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.
- Consider the following XML configuration for a bean definition:

```xml
<bean id="userCart"
      class="com.jediver.spring.view.dto.UserCart"
      scope="session" />
```

Bean Scopes

4. session (Lifecycle)



Bean Scopes

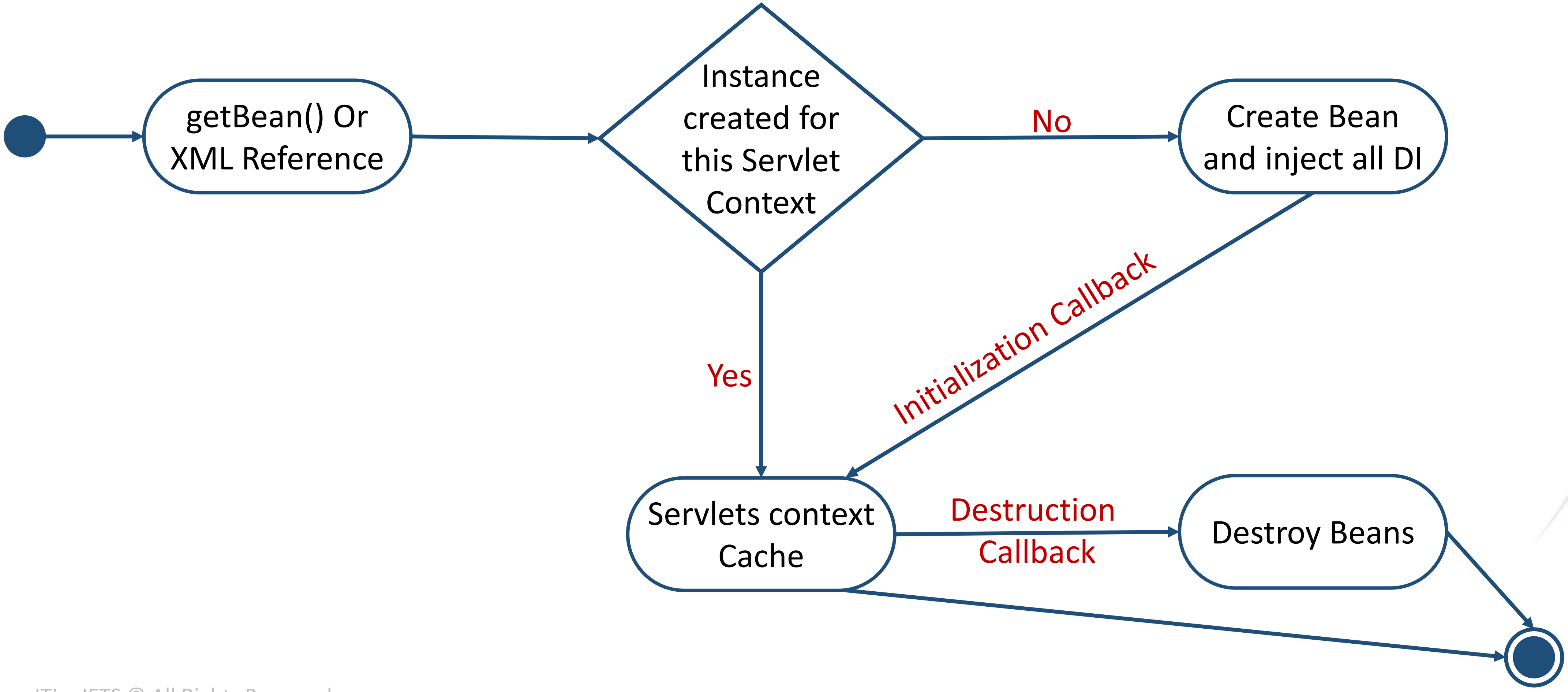
5. application

- The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application.
- That is, the appPreferences bean is scoped at the ServletContext level.
- This is somewhat similar to a Spring singleton bean but differs in two important ways:
  - It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application).
  - It is actually exposed and therefore visible as a ServletContext attribute.
- Consider the following XML configuration for a bean definition:

```
<bean id="appPreferences"
      class="com.jediver.spring.view.AppPreferences"
      scope="application"/>
```

Bean Scopes

5. application (Lifecycle)



Bean Scopes

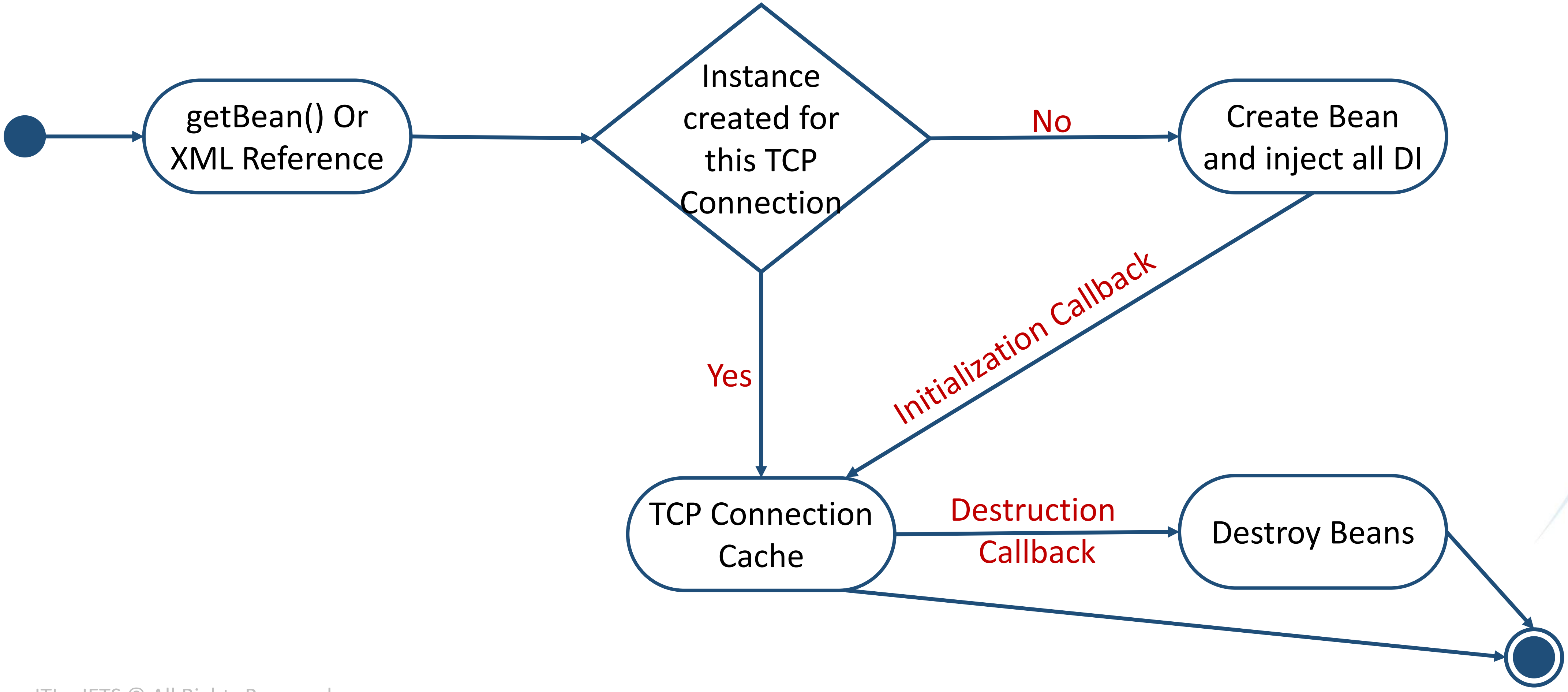
6. **websocket**

- The Spring container creates a new instance of the UserSession bean by using the userSession bean definition for the lifetime of a single TCP Connection **(Stateful per TCP Connection)**.
- When the TCP Connection is eventually discarded, the bean that is scoped to that particular websocket is also discarded.
- Consider the following XML configuration for a bean definition:

```xml
<bean id="userSession"
        class="com.jediver.spring.connection.UserSession"
        scope="websocket" />
```

Bean Scopes

6. **websocket (Lifecycle)**



Lazy-initialized Beans

- The default behavior for ApplicationContext
  - Implementations eagerly create and configure all singleton beans as part of the initialization process.
- When this Pre-instantiation is desirable:
  - Because errors in the configuration or surrounding environment are discovered immediately once the ApplicationContext created.
- When this Pre-instantiation is not desirable:
  - The time and memory needed to create these beans before they are actually needed.
- you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized.

Lazy-initialized Beans (Ex.)

- A lazily-initialized bean indicates to the IoC container to create a bean instance when it is first requested, rather than at startup.
- This behavior is controlled by the lazy-init attribute on the <bean/> element, as example:

```xml
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        lazy-init="true"/>
```

- You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element, a the following example shows:

```xml
<beans  default-lazy-init="true">
```

Lesson 8

Core Container

(Customizing the Nature of a Bean)

Lifecycle Callbacks

- To interact with the container's management of the bean lifecycle.
- To do this we have three ways:
  1. First use the interface
     - You can implement the Spring InitializingBean and DisposableBean interfaces.
  2. Use generic initialization and/or generic Destruction method
  3. Use Annotation
     - (@PostConstruct, @PreDestroy) annotation
- Internally, the Spring Framework uses BeanPostProcessor implementations to process any callback interfaces it can find and call the appropriate methods.

Initialization Callbacks

- We use Initialization callbacks to allow a bean to perform initialization work after all necessary properties on the bean have been set by the container.

- To do this we have three ways:

1. **First use the interface**
2. **Use generic initialization method**
3. **@PostConstruct annotation**

Initialization Callbacks

1. First use the interface

- The org.springframework.beans.factory.InitializingBean interface lets a bean perform initialization work after the container has set all necessary properties on the bean.
- The InitializingBean interface specifies a single method:
  - public void afterPropertiesSet() throws Exception;
- You make your bean implement InitializingBean interface that makes you implement afterPropertiesSet() and perform initialization work.

Initialization Callbacks

1. First use the interface

- Your Class:

```
public class UserDAO implements InitializingBean {

    @Override
    public void afterPropertiesSet() throws Exception {

    }
    {...}
}
```

- Bean Definition in XML:

```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"/>
```

Initialization Callbacks

1. First use the interface

    - Not recommend to use the InitializingBean interface, because it unnecessarily couples the code to Spring.
    - Alternatively:
        - We suggest using the @PostConstruct annotation
        - Or specifying a POJO initialization method.

Initialization Callbacks

2. Use generic initialization method

   - In the case of XML-based configuration metadata.
   - You can use the init-method attribute to specify the name of the method that has a void no-argument signature.
   - With Java configuration, you can use the initMethod attribute of @Bean.
   - It’s preferable to use @PostConstruct annotation instead of generic initialization method.

Initialization Callbacks

2. Use generic initialization method

- Your Class:

```java
public class UserDAO {

    public void init() {
        // do some initialization work
    }

    {...}
}
```

- Bean Definition in XML:

```xml
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      init-method="init"/>
```

Destruction callbacks

- We use destruction callbacks to allow a bean to perform destruction work before destroy the bean.

- To do this we have three ways:

1. **First use the interface**
2. **Use generic destruction method**
3. **@PreDestroy annotation**

Destruction callbacks

1. First use the interface

- The org.springframework.beans.factory.DisposableBean interface lets a bean perform destruction work before the container destroy the bean.
- The DisposableBean interface specifies a single method:
  - public void destroy() throws Exception;
- You make your bean implement DisposableBean interface that makes you implement destroy() and perform destruction work.

Destruction callbacks

1. First use the interface

- Your Class:

```java
public class UserDAO implements DisposableBean{

    @Override
    public void destroy() throws Exception {

    }

    {...}

}
```

- Bean Definition in XML:

```xml
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"/>
```

Destruction callbacks

1. First use the interface
   - Not recommend to use the DisposableBean interface, because it unnecessarily couples the code to Spring.
   - Alternatively:
     - We suggest using the @PreDestroy annotation
     - Or specifying a POJO destruction method.

Destruction callbacks

2. Use generic destruction method

- In the case of XML-based configuration metadata.
- You can use the destroy-method attribute to specify the name of the method that has a void no-argument signature.
- With Java configuration, you can use the destroyMethod attribute of @Bean.
- It's preferable to use @PreDestroy annotation instead of generic destruction method.

Destruction callbacks

2. Use generic destruction method

- Your Class:

```java
public class UserDAO {

    public void cleanup() {

    }
    {...}
}
```

- Bean Definition in XML:

```xml
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      destroy-method="cleanup"/>
```

Default Initialization and Destroy Methods

- If you write methods with names such as init(), initialize(), dispose(), and so on.
- Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.
- You can configure the Spring container to “look” for named initialization and destroy callback method names on every bean.
  - This means that you, as an application developer, can write your application classes and use an initialization callback called init(), without having to configure an init-method="init" attribute with each bean definition.
  - This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Default Initialization and Destroy Methods (Ex.)

```java
public class UserDAO {

    public void init() {

    }
    public void destroy() {

    }
    {...}
}
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-init-method="init"
       default-destroy-method="destroy">
```

- you can override the default by specifying (in XML, that is) the method name by using the init-method and destroy-method attributes of the <bean/> itself.

Combining Lifecycle Mechanisms

- You can combine these three mechanisms to control a given bean.
- With different methods:
  - Initialization methods, are called as follows:
    1. Methods annotated with @PostConstruct
    2. afterPropertiesSet() as defined by the InitializingBean callback interface.
    3. A custom configured init() method
  - Destruction methods, are called as follows:
    1. Methods annotated with @PreDestroy
    2. destroy() as defined by the DisposableBean callback interface
    3. A custom configured destroy() method

Combining Lifecycle Mechanisms (Ex.)

- With same methods:
  - However, if the same method name is configured for the different mechanism.
  - For example:
    - init() for an initialization method and also configured by @PostConstruct
    - that method is executed once.
  - Also Applied for the destruction.

Shutting Down the Spring IoC Container

- Shutting Down the Spring IoC Container Gracefully in Non-Web Applications
- Spring’s web-based ApplicationContext implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.
- If you use Spring’s IoC container in a non-web application environment (ex. in a rich client desktop environment), register a shutdown hook with the JVM.
    - Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.
- To register a shutdown hook, call the registerShutdownHook() method that is declared on the ConfigurableApplicationContext interface.

Shutting Down the Spring IoC Container (Ex.)

- Shutting Down the Spring IoC Container Gracefully in Non-Web Applications
- Spring’s web-based ApplicationContext implementations already have code in place to

```
public static void main(String[] args) {
    ConfigurableApplicationContext context
            = new ClassPathXmlApplicationContext("beans.xml");
    context.registerShutdownHook();
    ...
}
```

Bean Definition Inheritance

- 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.
- If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class.
- Most users do not work with them on this level. Instead, they configure bean definitions declaratively in a class such as the ClassPathXmlApplicationContext.
- When you use XML-based configuration metadata, you can indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

Bean Definition Inheritance (Ex.)

```java
public class Parent {

    private String name;
    private int age;

    public String getName() {...3 lines }

    public void setName(String name) {...3 lines }

    public int getAge() {...3 lines }

    public void setAge(int age) {...3 lines }

    @Override
    public String toString() {...3 lines }

}
```

```java
public class Child {

    private String name;
    private int age;
    private String address;

    public String getName() {...3 lines }

    public void setName(String name) {...3 lines }

    public int getAge() {...3 lines }

    public void setAge(int age) {...3 lines }

    public String getAddress() {...3 lines }

    public void setAddress(String address) {...3 lines }

    @Override
    public String toString() {...3 lines }

}
```

Bean Definition Inheritance (Ex.)

```xml
<bean id="parent" abstract="true"
        class="com.jediver.spring.core.bean.inheritance.Parent">
    <property name="name" value="Parent Name"/>
    <property name="age" value="20"/>
</bean>
```

```xml
<bean id="child" parent="parent"
        class="com.jediver.spring.core.bean.inheritance.Child">
    <property name="name" value="Child Name"/>
    <property name="address" value="Cairo"/>
</bean>
```

Bean Definition Inheritance (Ex.)

- A child bean definition uses the bean class from the parent definition if none is specified but can also override it.
- If the child bean class must be compatible with the parent:
  - It must accept the parent's property values.
- A child bean definition inherits from parent:
  - scope
  - constructor argument values
  - property values
  - initialization method
  - destroy method
  - static factory method

Bean Definition Inheritance (Ex.)

- A child bean definition can also override all inherited settings.
- The remaining settings are always taken from the child definition:
  - depends on
  - autowire mode
  - dependency check
  - singleton
  - lazy init.
- The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute.

Bean Definition Inheritance (Ex.)

- If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required.

```xml
<bean id="parent" abstract="true">
    <property name="name" value="Parent Name"/>
    <property name="age" value="20"/>
</bean>
```

```xml
<bean id="child" parent="parent"
        class="com.jediver.spring.core.bean.inheritance.Child">
    <property name="name" value="Child Name"/>
    <property name="address" value="Cairo"/>
</bean>
```

Collections

- The Collections are:
    - <list/> element.
    - <set/> element.
    - <map/> element.
    - <props/> element.
- Supports merging collections.
- If you merge in case of a parent can have <list/>, <map/>, <set/> or <props/> element and child have <list/>, <map/>, <set/> or <props/> elements inherit and override values from the parent collection.
- That is, the child collection's values are the result of merging the elements of the parent and child collections.

Collections <list/> element.

```xml
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent" abstract="true"
    class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails2">
        <list>
            <value>Hello World</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

```
Hello World
User{name=AhmedMedhat, balance=0.0}
Hello World 2
User{name=AhmedMedhat, balance=0.0}
```

```xml
<bean id="child" parent="parent">
    <property name="adminEmails2">
        <list merge="true">
            <value>Hello World 2</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

Collections <set/> element.

```xml
<bean id="user"
      class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent" abstract="true"
     class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails4">
        <set>
            <value>Hello World</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

```
Hello World
User{name=AhmedMedhat, balance=0.0}
Hello World 2
```

```xml
<bean id="child" parent="parent">
    <property name="adminEmails4">
        <set merge="true">
            <value>Hello World 2</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

Collections <map/> element.

```xml
<bean id="parent" abstract="true"
        class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails3">
        <map>
            <entry key="administrator" value="administrator@example.com"/>
            <entry key="support" value="support@example.com"/>
        </map>
    </property>
</bean>
```

```
administrator@example.com
support@example.co.uk
sales@example.com
```

```xml
<bean id="child" parent="parent">
    <property name="adminEmails3">
        <map merge="true">
            <entry key="sales" value="sales@example.com"/>
            <entry key="support" value="support@example.co.uk"/>
        </map>
    </property>
</bean>
```

Collections <props/> element.

```xml
<bean id="parent" abstract="true"
        class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
```

```
support@example.co.uk
administrator@example.com
sales@example.com
```

```xml
<bean id="child" parent="parent">
    <property name="adminEmails">
        <!-- the merge is specified on the child collection definition -->
        <props merge="true">
            <prop key="sales">sales@example.com</prop>
            <prop key="support">support@example.co.uk</prop>
        </props>
    </property>
</bean>
```

Collections (Ex.)

- This merging behavior applies similarly to the <list/>, <map/>, and <set/> collection types.
- In <list/> The parent’s values precede all of the child list’s values in .
- In <map/> & <set/>, no ordering exists.
- **Limitations of Collection Merging**
  - You cannot merge different collection types (such as a Map and a List). If you do attempt to do so, an appropriate Exception is thrown.
  - The merge attribute must be specified on the lower, inherited, child definition.
  - Specifying the merge attribute on a parent collection definition is redundant and does not result in the desired merging

PropertyPlaceholderConfigurer

- The PropertyPlaceholderConfigurer is used to externalize property values from an XML configuration file.
- Into another separate file in the standard Java Properties format.
- This is useful to allow the person deploying an application to customize environment-specific properties,
- without the risk of modifying the main XML definition file or files for the container.

PropertyPlaceholderConfigurer (Ex.)

- You can convert the text inside the <value> element into a Properties instance using the PropertyEditor.

```
<bean id="mappings"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
        <value>
            jdbc.driverClassName= com.mysql.jdbc.Driver
            jdbc.url= jdbc:mysql://localhost:3306/biddingschema
            jdbc.username= root
            jdbc.password= root
        </value>
    </property>
</bean>
```

PropertyPlaceholderConfigurer (Ex.)

- A DataSource with placeholder values is defined.
- It will configure some properties from an external Properties file, and at runtime.
- The actual values come from another file in the standard Java Properties format.

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/biddingschema
jdbc.username= root
jdbc.password= root
```

PropertyPlaceholderConfigurer (Ex.)

```xml
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>com/jediver/spring/core/cfg/jdbc.properties</value>
    </property>
</bean>
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

PropertyPlaceholderConfigurer (Ex.)

- You can create it in two way
  - Direct way with define bean definition for PropertyPlaceholderConfigurer class in container
  - With the context namespace introduced in Spring 2.5.

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd"
```

- Multiple locations may be provided as a comma-separated list for the location attribute.

```
<context:property-placeholder
            location="com/jediver/spring/core/cfg/jdbc.properties"/>
```

PropertyPlaceholderConfigurer (Ex.)

- The PropertyPlaceholderConfigurer look for properties in
  - The Properties file.
  - Checks the Java System properties.
    - If it cannot find a property you are trying to use.

PropertyPlaceholderConfigurer (Ex.)

- Set how to check system properties: as (fallback, override, or never).
- For example, will resolve ${user.dir} to the "user.dir" system property.
    - The default is "fallback":
        - If not being able to resolve a placeholder with the specified properties, a system property will be tried.
    - "override"
        - will check for a system property first, before trying the specified properties.
    - "never"
        - will not check system properties at all.

Lesson 9

Core Container (Annotation-based Configuration)

Annotations VS. XML for configuring Spring?

- The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML.
  - The short answer is "it depends".
  - The long answer is that each approach has its pros and cons, and usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined,
  - Annotations provide a lot of context in their declaration, leading to shorter and more concise configuration.
  - However, XML excels at wiring up components without touching their source code or recompiling them.
  - Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.
- No matter Your choice, Spring can accommodate both styles and even mix them together.

Annotation-based Configuration

- Spring 2.0 introduced
  - @Required annotation to enforce required properties.
- Spring 2.5 introduced
  - @Autowired annotation provides the capabilities of autowiring between dependencies as replacement to "autowire" attribute in xml but with more fine-grained control and wider applicability.
  - Also added support for JSR-250 annotations, such as
    - @PostConstruct and @PreDestroy.
- Spring 3.0 introduced
  - Added support for JSR-330 annotations (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named.

Annotation-based Configuration (Ex.)

- Use of these annotations also requires that certain specialized classes be registered within the Spring container.
- These can be registered as individual bean definitions:
  - RequiredAnnotationBeanPostProcessor.
  - AutowiredAnnotationBeanPostProcessor.
  - CommonAnnotationBeanPostProcessor.
  - PersistenceAnnotationBeanPostProcessor.
- OR implicitly registered by including the following tag in an XML-based
  - <annotation-config> in context namespace this implicitly registered all above BeanPostProcessor except RequiredAnnotationBeanPostProcessor.

Annotation-based Configuration (Ex.)

- These can be registered as individual bean definitions:
  - RequiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

  - AutowiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

  - CommonAnnotationBeanPostProcessor.

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>
```

  - PersistenceAnnotationBeanPostProcessor.

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
```

Annotation-based Configuration (Ex.)

- OR using context namespace introduced in Spring 2.5 :

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd"
```

- <annotation-config> Tag.

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

Annotation-based Configuration (Ex.)

- @Required these can be registered as bean for spring 5.1:
  - RequiredAnnotationBeanPostProcessor.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

  - This annotation indicates that the affected bean property must be populated at configuration time.
  - The @Required annotation applies to bean property setter methods, as in the following example:

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Required
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - @Autowired annotation provides the capabilities of autowiring but with more fine-grained control.
  - You can apply the @Autowired annotation to constructors.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public AccountServiceImpl(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

  - Starting from Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with.
  - Autowiring are applied by type.

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to property itself.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDAO accountDAO;
```

  - Also you don’t have to implement setter for this property
  - Autowiring are applied by type.

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to "traditional" setter methods.

```java
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

```xml
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to any setup methods.
  - These methods are called automatically called from spring container and autowire all parameters in these method.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public void setup(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;

    }
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - Also You can apply the @Autowired annotation to any setup methods.
  - These methods are called automatically called from spring container and autowire all parameters in these method.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;
    private Connection connection;

    @Autowired
    public void setup(AccountDAO accountDAO, Connection connection) {
        this.accountDAO = accountDAO;
        this.connection = connection;
    }
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects an array of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Account[] accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a List of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private List<Account> accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a Set of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Set<Account> accounts;
```

Annotation-based Configuration (Ex.)

- @Autowired :
  - You can also provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method.
  - Expects a Map of that type.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    private Map<String, Account> accounts;
```

  - By default, the autowiring fails whenever zero candidate beans are available.

Annotation-based Configuration (Ex.)

- The default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies.

- You can change this behavior by define required attribute.

```
public class AccountServiceImpl implements AccountService {

    @Autowired(required = false)
    private AccountDAO accountDAO;
```

Annotation-based Configuration (Ex.)

- Only one annotated constructor per-class can be marked as required, but multiple non-required constructors can be annotated.
  - In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied — that is, the constructor that has the largest number of arguments.
- Ex. Other constructors marked @Autowired(required=false).
- The required attribute of @Autowired is recommended over the @Required annotation.

Annotation-based Configuration (Ex.)

- Alternatively, you can express the non-required nature of a particular dependency through Java 8's java.util.Optional

```java
public class AccountServiceImpl implements AccountService {

    private Optional<AccountDAO> accountDAO;

    @Autowired
    public void setAccountDAO(Optional<AccountDAO> accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.get().addAccount(account);
    }
}
```

Annotation-based Configuration (Ex.)

- Starting from Spring Framework 5.0, you can also use a @Nullable annotation (of any kind in any package.
- Ex. jakarta.annotation.Nullable from JSR-305):

```java
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    public void setAccountDAO(@Nullable AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Annotation-based Configuration (Ex.)

- You can also use @Autowired for interfaces that are well-known resolvable dependencies:
    - BeanFactory
    - ApplicationContext
    - Environment
    - ResourceLoader
    - ApplicationEventPublisher
    - MessageSource.
- These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

Annotation-based Configuration (Ex.)

- The following example autowires an ApplicationContext object:

```
public class AccountDAOImpl implements AccountDAO {

    @Autowired
    private ApplicationContext context;
```

Annotation-based Configuration (Ex.)

- Since autowiring by type may lead to multiple candidates.
- It is necessary to have more control over the selection process.
  - You can do that by define primary attribute for bean definition.
- The other way is to accomplish this is with Spring's @Qualifier annotation.
- When you need more control over the selection process, you can use Spring’s @Qualifier annotation.

Annotation-based Configuration (Ex.)

- @Qualifier By id or name of bean definition:

```java
public class AccountServiceImpl implements AccountService {

    @Autowired
    @Qualifier("accountDao")
    private AccountDAO accountDAO;
```

- @Qualifier By qualifier value specified in bean definition:

```java
public class AccountServiceImpl implements AccountService {

    @Autowired
    @Qualifier("mainAccountDao")
    private AccountDAO accountDAO;
```

```xml
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
    <qualifier value="mainAccountDao"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Qualifier also applied for setup methods parameters:

```java
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Autowired
    @Qualifier("mainAccountDao")
    public void setup(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

```xml
<bean id="accountDao"
        class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
    <qualifier value="mainAccountDao"/>
</bean>
```

Annotation-based Configuration (Ex.)

- You may create your own custom qualifier annotations as well.
- Simply define an annotation and provide the @Qualifier annotation with your own attributes.

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface DAOQualifier {

    String name();

    Mobile mobile();

}
```

- Mobile is an enum:

```
public enum Mobile {
    Etisalat, Vodafone, Orange
}
```

Annotation-based Configuration (Ex.)

- You can refer now to your bean definition by your custom qualifier.

```
public class AccountServiceImpl implements AccountService {

    @Autowired
    @DAOQualifier(name = "ahmed", mobile = Mobile.Etisalat)
    private Account firstAccount;

    @Autowired
    @DAOQualifier(name = "ahmed", mobile = Mobile.Vodafone)
    private Account secondAccount;

    @Autowired
    @DAOQualifier(name = "mohamed", mobile = Mobile.Etisalat)
    private Account thirdAccount;
```

Annotation-based Configuration (Ex.)

```xml
<bean id="account1" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="ahmed"/>
        <attribute key="mobile" value="Etisalat"/>
    </qualifier>
</bean>
<bean id="account2" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="ahmed"/>
        <attribute key="mobile" value="Vodafone"/>
    </qualifier>
</bean>
<bean id="account3" class="com.jediver.spring.core.dal.entity.Account">
    <qualifier type="com.jediver.spring.core.dal.dao.DAOQualifier">
        <attribute key="name" value="mohamed"/>
        <attribute key="mobile" value="Etisalat"/>
    </qualifier>
</bean>
```

Annotation-based Configuration (Ex.)

- Spring also supports injection by using the JSR-250 @Resource annotation (jakarta.annotation.Resource) on fields or bean property setter methods.
- @Resource is a common pattern in Java EE (Ex. EJB, JSF-managed beans, JAX-WS endpoints).
- Also spring supports this pattern for Spring-managed objects as well.
- @Resource if a name specified,
  - Spring interprets that value as the bean name to be injected.

```
public class AccountServiceImpl implements AccountService {

    @Resource(name = "accountDao")
    private AccountDAO accountDAO;
```

Annotation-based Configuration (Ex.)

- @Resource :
  - Also You can apply the @Resource annotation to "traditional" setter methods.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Resource(name = "accountDao")
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

```
<bean id="accountDao"
      class="com.jediver.spring.core.dal.dao.impl.AccountDAOImpl">
    <constructor-arg ref="session"/>
</bean>
```

Annotation-based Configuration (Ex.)

- @Resource If no name is specified, the default name is derived from the field name or setter method.
  - In case of a field, it takes the field name.
  - In case of a setter method, it takes the bean property name.

```java
public class AccountServiceImpl implements AccountService {

    @Resource
    private AccountDAO accountDAO;
```

- @Resource If no name is specified and didn’t find match for fieldName or proertyName, then finds a primary type match instead of a specific named bean.

Annotation-based Configuration (Ex.)

- You can also use @Resource for interfaces that are well-known resolvable dependencies:
  - BeanFactory
  - ApplicationContext
  - Environment
  - ResourceLoader
  - ApplicationEventPublisher
  - MessageSource.
- These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

Annotation-based Configuration (Ex.)

- The following example autowires an ApplicationContext object by @Resource:

```
public class AccountDAOImpl implements AccountDAO {

    @Resource
    private ApplicationContext context;
```

Annotation-based Configuration (Ex.)

- The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations:
  - jakarta.annotation.PostConstruct
  - jakarta.annotation.PreDestroy.
- Introduced in Spring 2.5, the support for these annotations offers an alternative to the lifecycle callback mechanism described in initialization callbacks and destruction callbacks.

Annotation-based Configuration (Ex.)

```
public class AccountDAOImpl implements AccountDAO {

    @PostConstruct
    public void init() {
        System.out.println(session.isConnected());
    }

    @PreDestroy
    public void destroy() {
        session.close();
    }
```

- Like @Resource, the @PostConstruct and @PreDestroy annotation types were a part of the standard Java libraries from JDK 6 to 8.
- However, the entire jakarta.annotation package got separated from the core Java modules in JDK 9 and eventually removed in JDK 11. If needed, the jakarta.annotation-api artifact needs to be obtained via Maven Central now.

Lesson 10

Core Container

(Classpath Scanning and Managed Components)

Classpath Scanning and Managed Components

- All previous examples in lessons use XML to specify the configuration metadata that produces each BeanDefinition within the Spring container.
- The previous lesson (Annotation-based Configuration) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file.
- This lesson describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container.
- This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, @Component).

Classpath Scanning and Managed Components (Ex.)

- Started from Spring 2.5
- Spring provides stereotype annotations:
  - @Component is a generic stereotype for any Spring-managed component.
  - @Repository is specialized for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).
  - @Service is specialized for service class.
  - @Controller is specialized for presentation layers.
- Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects.

Classpath Scanning and Managed Components (Ex.)

- Using context namespace introduced in Spring 2.5 :

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd"
```

- Instead of using <annotation-config> Tag.

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

- You use <context:component-scan> Tag and it automatically use <annotation-config> Tag internally.

```
<context:component-scan base-package="com.jediver.spring.core"/>
```

Classpath Scanning and Managed Components (Ex.)

- Using context namespace introduced in Spring 2.5 :

```
@Repository
public class AccountDAOImpl implements AccountDAO {

    @PostConstruct
    public void init() {
        System.out.println(session.isConnected());
    }

    @PreDestroy
    public void destroy() {
        session.close();
    }
```

```
@Service
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Resource
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Classpath Scanning and Managed Components (Ex.)

- By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components.
- However, you can modify and extend this behavior by applying custom filters.
  - Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation.
  - Or as include-filter or exclude-filter child elements of the component-scan element.

Classpath Scanning and Managed Components (Ex.)

Filter Type	Example Expression	Description
annotation (default)	org.example.SomeAnnotation	An annotation to be present at the type level in target components.
assignable	org.example.SomeClass	A class (or interface) that the target components are assignable to (extend or implement).
aspectj	org.example..*Service+	An AspectJ type expression to be matched by the target components.
regex	org\.example\.Default.*	A regex expression to be matched by the target components class names.
custom	org.example.MyTypeFilter	A custom implementation of the org.springframework.core.type .TypeFilter interface.

Classpath Scanning and Managed Components (Ex.)

- The following example shows the configuration ignoring all @Repository annotations and using "stub" repositories instead:

```xml
<context:component-scan base-package="com.jediver.spring.core">
    <context:include-filter type="regex"
            expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

Classpath Scanning and Managed Components (Ex.)

- It is also possible to disable the default filters
  - by providing use-default-filters="false" as an attribute of the <component-scan/> element.
- This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, or @Controller.

```
<context:component-scan base-package="com.jediver.spring.core"
                        use-default-filters="false"/>
```

Naming autodetected components

- When a component is autodetected as part of the scanning process.
- If 'stereotype' annotation (@Component, … etc) contains a name value, corresponding bean will have that name
- If not, the bean name will be the un-capitalized non-qualified class name.
- Example,
    - if the class name is : Employee
    - Bean name will be : employee

```
@Repository("accountDAO")
public class AccountDAOImpl implements AccountDAO {
```

Autodetected Components Scope

- The default scope is 'singleton'.
- However, there are times when other scopes are needed.
- Therefore Spring 2.5 introduces a new @Scope annotation as well.
- Simply provide the name of the scope within the annotation, such as:

```
@Scope("prototype")
@Repository("accountDAO")
public class AccountDAOImpl implements AccountDAO {
```

Autodetected Components Scope

- Note:
  - If you would like to provide a custom strategy for scope resolution rather than relying on the annotation-based approach.
  - Implement the ScopeMetadataResolver interface, and be sure to include a default no-arg constructor.
  - Then, provide the fully-qualified class name when configuring the scanner:

```java
public class CustomScope implements ScopeMetadataResolver {

    @Override
    public ScopeMetadata resolveScopeMetadata(BeanDefinition bd) {
        //do your logic here and return the scope metadata
        return null;
    }
}
```

```xml
<context:component-scan base-package="com.jediver.spring.core"
    scope-resolver="com.jediver.spring.core.dal.cfg.CustomScope"/>
```

Lesson 11

Core Container

(Using JSR 330 Standard Annotations)

Using JSR 330 Standard Annotations

- Introduced in Spring 3.0,
- Spring offers support for JSR-330 standard annotations (Dependency Injection).
- Those annotations are scanned in the same way as the Spring annotations.
- To use them, you need to have the relevant jars in your classpath.

```
<dependency>
        <groupId>jakarta.inject</groupId>
        <artifactId>jakarta.inject-api</artifactId>
        <version>2.0.1</version>
 </dependency>
```

Dependency Injection with @Inject and @Named

- Instead of @Autowired, you can use @jakrta.inject.Inject as follows:

```java
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }
```

Dependency Injection with @Inject and @Named (Ex.)

- As with @Autowired, you can use @Inject at
  - The field level
  - The method level
  - The constructor-argument level.
- Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call.

Dependency Injection with @Inject and @Named (Ex.)

- As the Following Example:

```java
public class AccountServiceImpl implements AccountService {

    private Provider<AccountDAO> accountDAO;

    @Inject
    public void setAccountDAO(Provider<AccountDAO> accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.get().addAccount(account);
    }
```

Dependency Injection with @Inject and @Named (Ex.)

- If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(@Named("aaa") AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.addAccount(account);
    }
```

Dependency Injection with @Inject and @Named (Ex.)

- As with @Autowired, @Inject can also be used with java.util.Optional.
- This is even more applicable here, since @Inject does not have a required attribute.

```
public class AccountServiceImpl implements AccountService {

    private Optional<AccountDAO> accountDAO;

    @Inject
    public void setAccountDAO(Optional<AccountDAO> accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.get().addAccount(account);
    }
```

Dependency Injection with @Inject and @Named (Ex.)

- As with @Autowired, @Inject can also be used with @Nullable.
- This is even more applicable here, since @Inject does not have a required attribute.

```
public class AccountServiceImpl implements AccountService {

    private AccountDAO accountDAO;

    @Inject
    public void setAccountDAO(@Nullable AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    @Override
    public void addAccount(Account account) {
        accountDAO.addAccount(account);
    }
```

@Named and @ManagedBean

- Standard Equivalents to the @Component Annotation
- Instead of @Component, you can use @jakrta.inject.Named or jakarta.annotation.ManagedBean.

```
@Named("accountDao")
public class AccountDAOImpl implements AccountDAO {
```

```
@ManagedBean("accountDao")
public class AccountDAOImpl implements AccountDAO {
```

Limitations of JSR-330 Standard Annotations

Spring	jakrta.inject.*	jakrta.inject restrictions / comments
@Autowired	@Inject	• @Inject has no 'required' attribute. • Can be used with Java 8’s Optional instead.
@Component	@Named / @ManagedBean	• JSR-330 does not provide a composable model • Only a way to identify named components.
@Scope("singleton")	@Singleton	• The JSR-330 default scope is like Spring’s prototype. • In order to use a scope other than singleton, you should use Spring’s @Scope annotation. • javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.

Limitations of JSR-330 Standard Annotations

Spring	jakrta.inject.*	jakrta.inject restrictions / comments
@Qualifier	@Qualifier / @Named	• javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. • Concrete String qualifiers (like Spring's @Qualifier with a value) can be associated through javax.inject.Named.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	• javax.inject.Provider is a direct alternative to Spring's ObjectFactory, only with a shorter get() method name. • It can also be used in combination with Spring's @Autowired or with non-annotated constructors and setter methods.

Lesson 12

Core Container
(Java-based Container Configuration)

Java-based Container Configuration

- Spring's new Java-configuration support are @Configuration-annotated classes and @Bean-annotated methods.
- Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions.
- You can define the @Configuration annotation on your configuration class/classes by

```
@Configuration
public class AppConfig {
}
```

```
@Configuration
public class ModelConfig {
```

Java-based Container Configuration

- Introduced in Spring 3.0.
- Spring's org.springframework.context.annotation.AnnotationConfigApplicationContext is capable of accepting not only @Configuration classes as input but also plain @Component classes and classes annotated with JSR-330 metadata.
- When @Configuration class/classes are provided as input.
  - The @Configuration class itself is registered as a bean definition.
  - And all declared @Bean methods within the class are also registered as bean definitions.
- When @Component and JSR-330 classes are provided
  - They are registered as bean definitions.
  - And it is assumed that DI metadata such as @Autowired or @Inject are used where necessary.

Java-based Container Configuration

- Like XML-Based style:
  - Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext.
- Annotation-Based style:
  - You can use @Configuration classes as input when instantiating an AnnotationConfigApplicationContext.
  - This allows for completely XML-free usage of the Spring container.
- The following example shows:

```
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);
```

Java-based Container Configuration

- You can instantiate an AnnotationConfigApplicationContext by using a no-arg constructor and then configure it by using the register() method.
- This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext. The following example shows how to do so:

```
AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();

context.register(ModelConfig.class);
context.register(AppConfig.class);
context.refresh();
```

- Or by dynamic args of AnnotationConfigApplicationContext class:

```
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class, ModelConfig.class);
```

Java-based Container Configuration

- In Earlier Versions, AnnotationConfigApplicationContext is working only with @Configuration classes. So you can configure as the following example shows:

```
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class,
            AccountDAOImpl.class, AccountServiceImpl.class);
```

Enabling Component Scanning

- AnnotationConfigApplicationContext exposes the scan(String...) method to allow component scanning functionality.
- as follows:

```
AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();
context.register(ModelConfig.class);
context.register(AppConfig.class);
context.scan("com.jediver.spring.core");
context.refresh();
```

Enabling Component Scanning

- Or fully annotation-based configuration.
- To enable component scanning, you can annotate your @Configuration class by @ComponentScan(basePackages = "")
- As follows:

```
@Configuration
@ComponentScan(basePackages = "com.jediver.spring.core")
public class AppConfig {

}
```

Using the @Bean Annotation

- For those familiar with Spring's <beans/> XML configuration.
  - The @Bean annotation plays the same role as the <bean/> element.
- @Bean is a method-level annotation is used to indicate that a method
  - instantiates, configures, and initializes a new object to be managed by the Spring IoC container.
- The annotation supports some of the attributes offered by <bean/>, such as:
  - name
  - init-method
  - destroy-method
  - ~~autowire~~ is become deprecated for Spring 5
  - autowireCandidate

Using the @Bean Annotation

- To declare a bean in annotation configuration
  - Either by using the @Bean annotation in a @Configuration
  - Or by using the @Component annotation in your POJO class.
- You can use @Bean methods
  - Either inside @Configuration Class
  - Or within any Spring @Component.
  - However, they are most often used with @Configuration beans (Recommended).
- You use this method to register a bean definition within an ApplicationContext of the type specified as the method's return value.
  - By default, the bean name is the same as the method name.

Using the @Bean Annotation

- The following example shows a @Bean method declaration:

```java
@Configuration
public class AppConfig {

    @Bean
    public Account myAccount() {
        return new Account();
    }
}
```

- You can look up for this bean by:

```java
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);
Account account = context.getBean(Account.class);
System.out.println(account);
```

```java
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);
Account account = (Account) context.getBean("myAccount");
System.out.println(account);
```

Using the @Bean Annotation (Ex.)

- You can be override default behavior with the name attribute.

```java
@Bean(name = "account1")
public Account myAccount() {
    return new Account();
}
```

- You can look up for this bean by:

```java
ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);
Account account = (Account) context.getBean("account1");
System.out.println(account);
```

Using the @Bean Annotation (Ex.)

- As discussed before, it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing.
- The name attribute of the @Bean annotation accepts a String array for this purpose.

```java
@Bean({"account1", "myAccount", "userAccount"})
public Account myAccount() {
    return new Account();
}
```

- You can look up for this bean by:

```java
ApplicationContext context
            = new AnnotationConfigApplicationContext(AppConfig.class);
Account account1 = (Account) context.getBean("account1");
Account account2 = (Account) context.getBean("myAccount");
Account account3 = (Account) context.getBean("userAccount");
System.out.println(account1 == account2);
System.out.println(account2 == account3);
```

@Bean Annotation Dependencies

- A @Bean-annotated method can have a number of parameters that describe the dependencies required to build that bean.
- For instance, if our AccountDAO requires a Session, we can materialize that dependency with a method parameter, as the following example shows:

```
@Bean
public AccountDAO accountDao(Session session) {
    return new AccountDAOImpl(session);
}
```

@Bean Annotation Callbacks

- Receiving Lifecycle Callbacks
  - initialization method within your Class

```java
public class UserDAO {

    public void init() {
        // do some initialization work
    }

    {...}
}
```

  - Bean Definition using annotation:

```java
@Bean(initMethod = "init")
public UserDAO userDao(Session session) {
    return new UserDAOImpl(session);
}
```

@Bean Annotation Callbacks (Ex.)

- Receiving Lifecycle Callbacks
  - Instead of using initMethod as attribute in @Bean as declared previously.

```
@Bean
public UserDAO userDao(Session session) {
    UserDAOImpl userDAO = new UserDAOImpl(session);
    userDAO.init();
    return userDAO;
}
```

@Bean Annotation Callbacks (Ex.)

- Receiving Lifecycle Callbacks
  - destroy method within your Class

```
public class UserDAO {

    public void cleanup() {

    }
    {...}
}
```

  - Bean Definition using annotation:

```
@Bean(destroyMethod = "cleanup")
public UserDAO userDao(Session session) {
    return new UserDAOImpl(session);
}
```

@Bean Annotation Callbacks (Ex.)

- By default, beans defined with Java configuration:
  - If you have a public close or shutdown method are automatically marked for a destruction callback.
  - If you have a public close or shutdown method and you do not wish for it to be called when the container shuts down.
    - You can add @Bean(destroyMethod="") to your bean definition to disable the default mode.
- You may want to do that by default for a resource that you acquire with JNDI, as its lifecycle is managed outside the application.
- In particular, make sure to always do it for a DataSource, as it is known to be problematic on Java EE application servers.

```
@Bean(destroyMethod = "")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

@Bean Annotation Scope

- Spring includes the @Scope annotation so that you can specify the scope of a bean.
- Using the @Scope Annotation
  - You can specify that your beans defined with the @Bean annotation should have a specific scope.
  - You can use any of the standard scopes specified before.
- The default scope is singleton, but you can override this with the @Scope annotation
- As follows:

```
@Bean
@Scope("prototype")
public AccountService accountService() {
    return new AccountServiceImpl();
}
```

@Configuration Annotation
Injecting Inter-bean Dependencies

- @Configuration is a class-level annotation indicating that an object is a source of bean definitions.
- @Configuration classes declare beans through public @Bean annotated methods.
- Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies.
- Injecting Inter-bean Dependencies
  - When beans have dependencies on one another.
  - Expressing that dependency is as simple as having one bean method call another.

@Configuration Annotation
Injecting Inter-bean Dependencies

- Injecting Inter-bean Dependencies

```
@Configuration
public class AppConfig {

    @Bean
    public AccountDAO accountDao() {
        return new AccountDAOImpl();
    }

    @Bean
    public AccountService accountService1() {
        return new AccountServiceImpl(accountDao());
    }
}
```

Using the @Import Annotation

- Like the <import/> element is used within Spring XML files to aid in modularizing configurations.
- The @Import annotation allows for loading @Bean definitions from another configuration class.
- Introduced in Spring 4.2
- As follows:

```
@Configuration
@Import(AppConfig1.class)
public class AppConfig {

    @Bean
    public AccountDAO accountDao() {
        return new AccountDAOImpl();
    }

}
```

```
@Configuration
public class AppConfig1 {

    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl();
    }

}
```

Using the @Import Annotation

- Constructor injection in @Configuration classes is only supported as of Spring Framework 4.3.
- As follows:

```java
@Configuration
@Import(AppConfig1.class)
public class AppConfig {

    @Bean
    public AccountDAO accountDao() {
        return new AccountDAOImpl();
    }

}
```

```java
@Configuration
public class AppConfig1 {

    @Bean
    public AccountService accountService(
            AccountDAO accountDao) {
        return new AccountServiceImpl(accountDao);
    }

}
```

Using the @Import Annotation

- Note also that there is no need to specify @Autowired if the target bean defines only one constructor.
- As follows:

```java
@Configuration
@Import(AppConfig1.class)
public class AppConfig {

    @Bean
    public AccountDAO accountDao() {
        return new AccountDAOImpl();
    }

}
```

```java
@Configuration
public class AppConfig1 {

    @Autowired
    private AccountDAO accountDao;

    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(accountDao);
    }

}
```

Using the @Import Annotation

- Fully-qualifying imported beans for ease of navigation
- For Example: If there is an accountDao instances from different configuration files
  - What version should spring will autowire ???
  - So we use Fully-qualifying imported beans.

```java
@Configuration
@Import(AppConfig1.class)
public class AppConfig {

    @Bean
    public AccountDAO accountDao() {
        return new AccountDAOImpl();
    }

}
```

```java
@Configuration
public class AppConfig1 {

    @Autowired
    private AppConfig appConfig;

    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(
                appConfig.accountDao());
    }
}
```

Combining Java and XML Configuration

- Spring’s @Configuration class support does not aim to be a 100% complete replacement for Spring XML.
- Some facilities, such as Spring XML namespaces, remain an ideal way to configure the container.
- In cases where XML is convenient or necessary, you have a choice:
  - (XML Definitions) Either instantiate the container in an "XML-centric" way by using for example ClassPathXmlApplicationContext.
  - (Annotation Definitions) Or instantiate it in a "Java-centric" way by using AnnotationConfigApplicationContext
  - (XML & Annotation Definitions) Or instantiate it in a "Java-centric" way by using AnnotationConfigApplicationContext and use the @ImportResource annotation to import XML as needed.

Combining Java and XML Configuration

- In applications where @Configuration classes are the primary mechanism for configuring the container, it is still likely necessary to use at least some XML.
- In these scenarios, you can use @ImportResource and define only as much XML as you need.
- The following example shows how to use the @ImportResource annotation:

```
@Configuration
@ImportResource("classpath:/com/jediver/spring/core/cfg/beans.xml")
public class AppConfig {
```

Combining Java and XML Configuration

```java
@Configuration
@ImportResource("classpath:/com/jediver/spring/core/cfg/beans.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}
```

```xml
<context:property-placeholder
    location="classpath:/com/jediver/spring/core/cfg/jdbc.properties"/>
```

```
jdbc.driverClassName= com.mysql.jdbc.Driver
jdbc.url= jdbc:mysql://localhost:3306/biddingschema
jdbc.username= root
jdbc.password= root
```

Bean Definition Profiles

- Bean definition profiles provide a mechanism in the core container that allows for registration of different beans in different environments.
- The word, "environment", can mean different things to different users, and this feature can help with many use cases.
  - For Example: Working against an in-memory datasource in development versus looking up that same datasource from JNDI when in QA or production.

Bean Definition Profiles (Ex.)

- The problem is how to switch between using these two variations based on the current environment.
- Over time, Spring users have devised a number of ways to get this done.
  - Relying on a combination of system environment variables.
  - XML <import/> statements containing ${placeholder} tokens that resolve to the correct configuration file path depending on the value of an environment variable.
- Bean definition profiles is a core container feature that provides a solution to this problem.

Bean Definition Profiles (Ex.)

- Using @Profile
- The @Profile annotation lets you indicate that a component is eligible for registration when one or more specified profiles are active.
- The Following configuration for production profile

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod = "")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

Bean Definition Profiles (Ex.)

- The Following configuration for development profile

```
@Configuration
@Profile("development")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/jediver/spring/core/dal/cfg/sql/schema.sql")
                .addScript("classpath:com/jediver/spring/core/dal/cfg/sql/data.sql")
                .build();
    }
}
```

Bean Definition Profiles (Ex.)

- @Profile("")
- The profile string may contain a simple profile name.
  - For Example, "production", "development"
- Or a profile expression.
- A profile expression allows for more complicated profile logic to be expressed
  - For Example, "production & us-east".

Bean Definition Profiles (Ex.)

- The following operators are supported in profile expressions:
  - ! A logical "not" of the profile
  - & A logical "and" of the profiles
  - | A logical "or" of the profiles
- You cannot mix the & and | operators without using parentheses.
  - For example, "production & us-east | eu-central" is not a valid expression.
  - It must be expressed as "production & (us-east | eu-central)"

Bean Definition Profiles (Ex.)

- You can use @Profile as a meta-annotation for the purpose of creating a custom composed annotation.
- The following example defines a custom @Production annotation that you can use as a drop-in replacement for @Profile("production"):

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

```
@Configuration
@Production
public class JndiDataConfig {

    @Bean(destroyMethod = "")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.
                lookup("java:comp/env/jdbc/datasource");
    }
}
```

Bean Definition Profiles (Ex.)

- @Profile can also be declared at the method level to include only one particular bean

```
@Configuration
public class AppConfig {

    @Bean(destroyMethod = "")
    @Production
    public DataSource dataSource1() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.
                lookup("java:comp/env/jdbc/datasource");
    }
```

```
    @Bean
    @Profile("development")
    public DataSource dataSource2() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/jediver/spring/core/dal/cfg/sql/schema.sql")
                .addScript("classpath:com/jediver/spring/core/dal/cfg/sql/data.sql")
                .build();
    }
}
```

Bean Definition Profiles (Ex.)

- XML Bean Definition Profiles
- You can configure Profile in XML by profile attribute of the <beans> element in different configuration files.
- As follows:

```xml
<beans profile="development"
        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">
```

Bean Definition Profiles (Ex.)

- You can configure Profile in XML by profile attribute of the <beans> element in same configuration file.
- As follows:

```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">
    <beans profile="development">

    </beans>
    <beans profile="production">

    </beans>
</beans>
```

Bean Definition Profiles (Ex.)

- Activating a Profile:
- Now that we have updated our configuration, we still need to instruct Spring which profile is active.
- If we started our sample application right now,
  - We would see a NoUniqueBeanDefinitionException: No qualifying bean of type 'javax.sql.DataSource' available: expected single matching bean but found 2: dataSource1, dataSource2

Bean Definition Profiles (Ex.)

- Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the Environment API which is available through an ApplicationContext.
- As Follows:

```
AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();
context.getEnvironment().setActiveProfiles("production");
context.register(AppConfig.class);
context.refresh();
```

PropertySource Abstraction

- Spring provide a high-level way of asking whether the my-property property is defined for the current environment.
- To answer this question, the Environment object performs a search over a set of PropertySource objects.
- A PropertySource is a simple abstraction over any source of key-value pairs, and Spring's StandardEnvironment is configured with two PropertySource objects:
  - One representing the set of JVM system properties (System.getProperties())
  - One representing the set of system environment variables (System.getenv()).

PropertySource Abstraction (Ex.)

- In the following Example: We ask for if PropertySource objects contains "JAVA_HOME" variable or not in runtime.

```
AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();
Environment env = context.getEnvironment();
boolean containsMyProperty = env.containsProperty("JAVA_HOME");
System.out.println("Does environment contain the 'JAVA_HOME'? "
        + containsMyProperty);
```

PropertySource Abstraction (Ex.)

- The search performed is hierarchical.
- In non Web-aware Application By default,
  1. JVM system properties
     - (-D command-line arguments)
  2. JVM system environment
     - (operating system environment variables)
- So, if the "JAVA_HOME" property happens to be set in both places during a call to env.getProperty("JAVA_HOME"), the system property value “wins” and is returned.
- Note:
  - That property values are not merged but rather completely overridden by a preceding entry.

PropertySource Abstraction (Ex.)

- In Web-aware Application By default,
  1. ServletConfig parameters
     - (if applicable — for example, in case of a DispatcherServlet context)
  2. ServletContext parameters
     - (web.xml context-param entries)
  3. JNDI environment variables
     - (java:comp/env/ entries)
  4. JVM system properties
     - (-D command-line arguments)
  5. JVM system environment
     - (operating system environment variables)

PropertySource Abstraction (Ex.)

- Most importantly, the entire mechanism is configurable.
- Perhaps you have a custom source of properties that you want to integrate into this search.

  To do so, implement and instantiate your own PropertySource and add it to the set of

  PropertySources for the current Environment.

PropertySource Abstraction (Ex.)

- The Following Example of custom PropertySource:

```java
public class MyPropertySource extends PropertySource {

    public MyPropertySource(String name) {
        super(name);
    }

    @Override
    public Object getProperty(String propertyName) {
        Object result = null;
        if (propertyName.equals("userId")) {
            result = "JEDiver";
        }
        return result;
    }

}
```

PropertySource Abstraction (Ex.)

- The Following Example of registering custom PropertySource into Spring's application context:

```
AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();
MutablePropertySources sources = context.getEnvironment()
        .getPropertySources();
sources.addFirst(new MyPropertySource("myresource"));
Environment env = context.getEnvironment();
String propertyValue = env.getProperty("userId");
System.out.println("userId is " + propertyValue);
```

Using @PropertySource

- The @PropertySource annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring's Environment.
- Given a file called jdbc.properties that contains the key-value pair.

```java
@Configuration
@PropertySource("classpath:/com/jediver/spring/core/cfg/jdbc.properties")
public class AppConfig {

    @Autowired
    Environment environment;

    @Bean
    public DataSource dataSource() {
        String url = environment.getProperty("jdbc.url");
        String username = environment.getProperty("jdbc.username");
        String password = environment.getProperty("jdbc.password");
        return new DriverManagerDataSource(url, username, password);
    }
}
```

Using @PropertySource

- Or by @Value annotation.

```
@Configuration
@PropertySource("classpath:/com/jediver/spring/core/cfg/jdbc.properties")
public class AppConfig {


    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}
```

References & Recommended Reading

References & Recommended Reading

- [Spring Framework Documentation Version 5.1.6.RELEASE]
- Spring in Action 5th Edition
- Cloud Native Java
- Learning Spring Boot 2.0
- Spring 5 Recipes: A Problem-Solution Approach