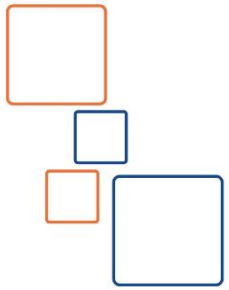
Spring Framework

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?

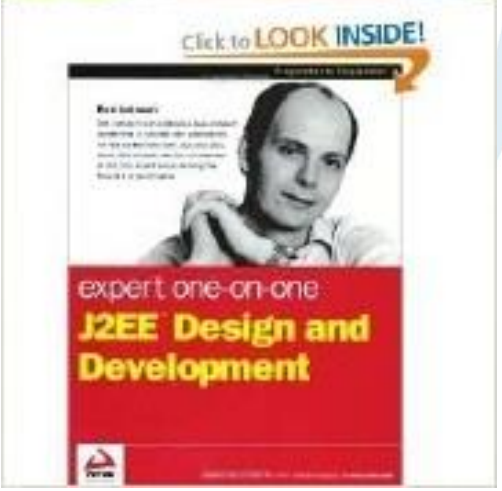
- Different packaging technique:
  - In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control.
  - Others may run as a single jar with the server embedded, possibly in a cloud environment.
  - Others 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.

Lesson 3

Spring Framework introduction

Java™ Education and Technology Services

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.

Java™ Education and Technology Services

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

```
public class HelloWorld {

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

- Problems:
  - 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

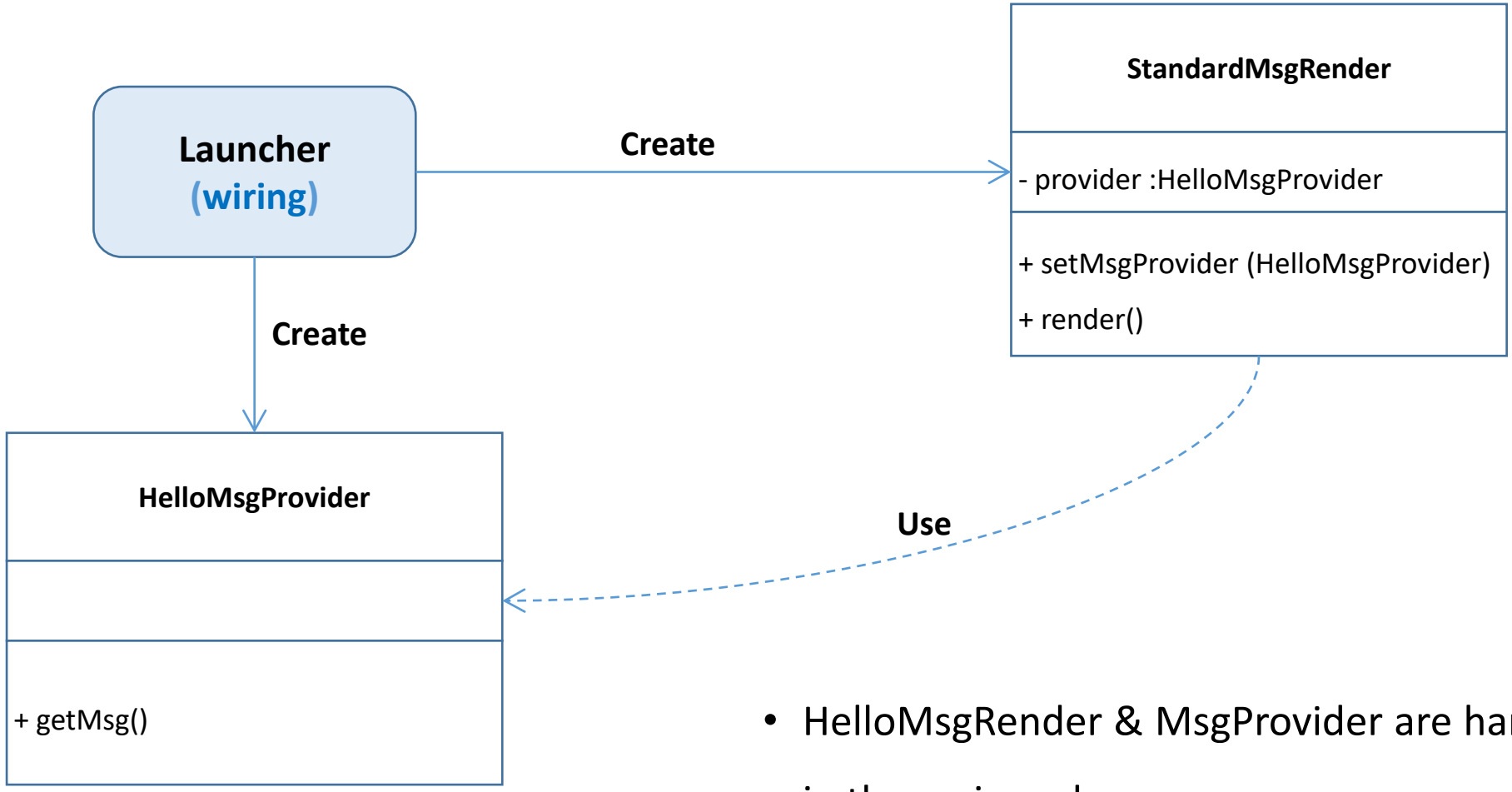
```
public class HelloWorld {

    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.)

- **Message provider logic:**
  - responsible for providing the message.
- **Message rendering logic:**
  - responsible for rendering the message.
- **Launcher:**
  - 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";
    }
}
```

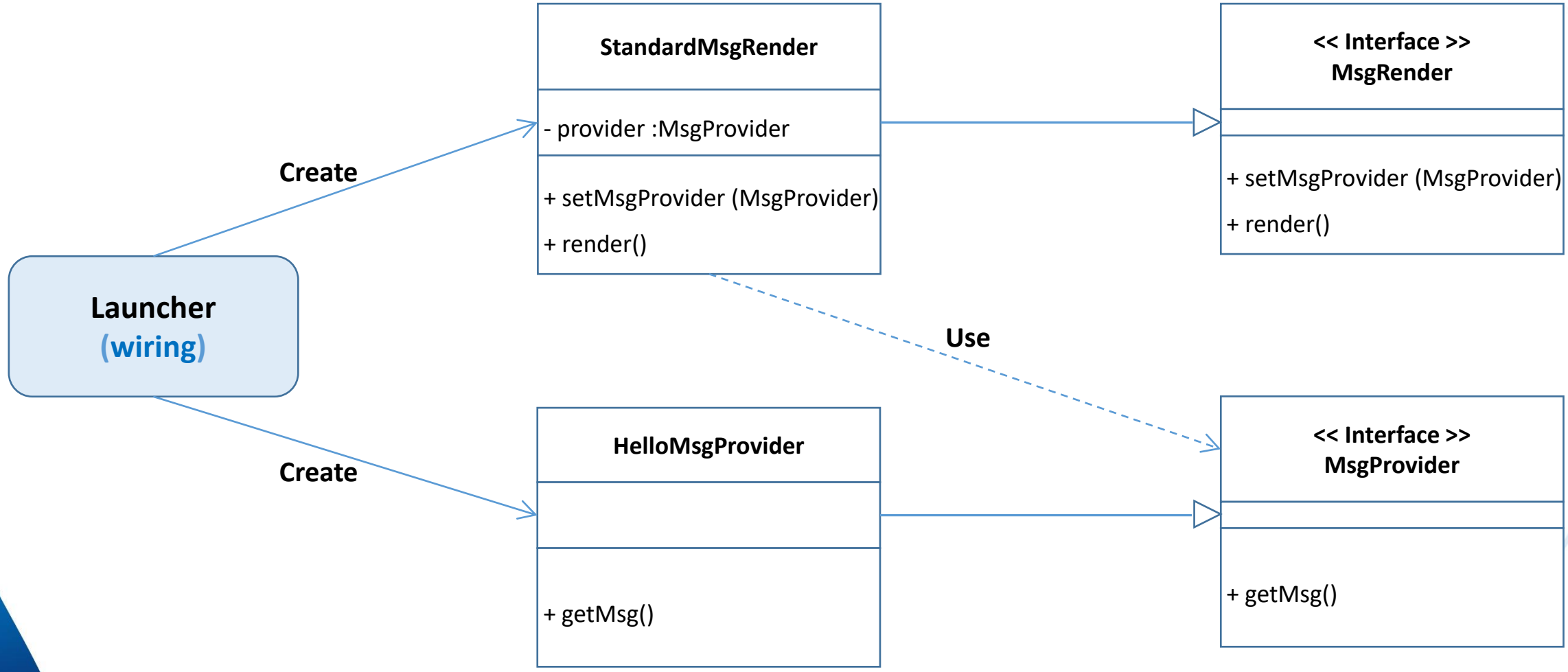
```java
public class StandardMsgRender {

    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.)

- **Message provider logic:**
  - Define the methods in an interface and the class that implements this interface.
  - Separated from Message Renderer.
- **Message rendering logic:**
  - Define the methods in an interface and the class that implements this interface.
  - Separated from Message Provider.
- **Launcher:**
  - 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.)

```
public interface MsgProvider {

    public String getMsg();
}
```

```
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.)

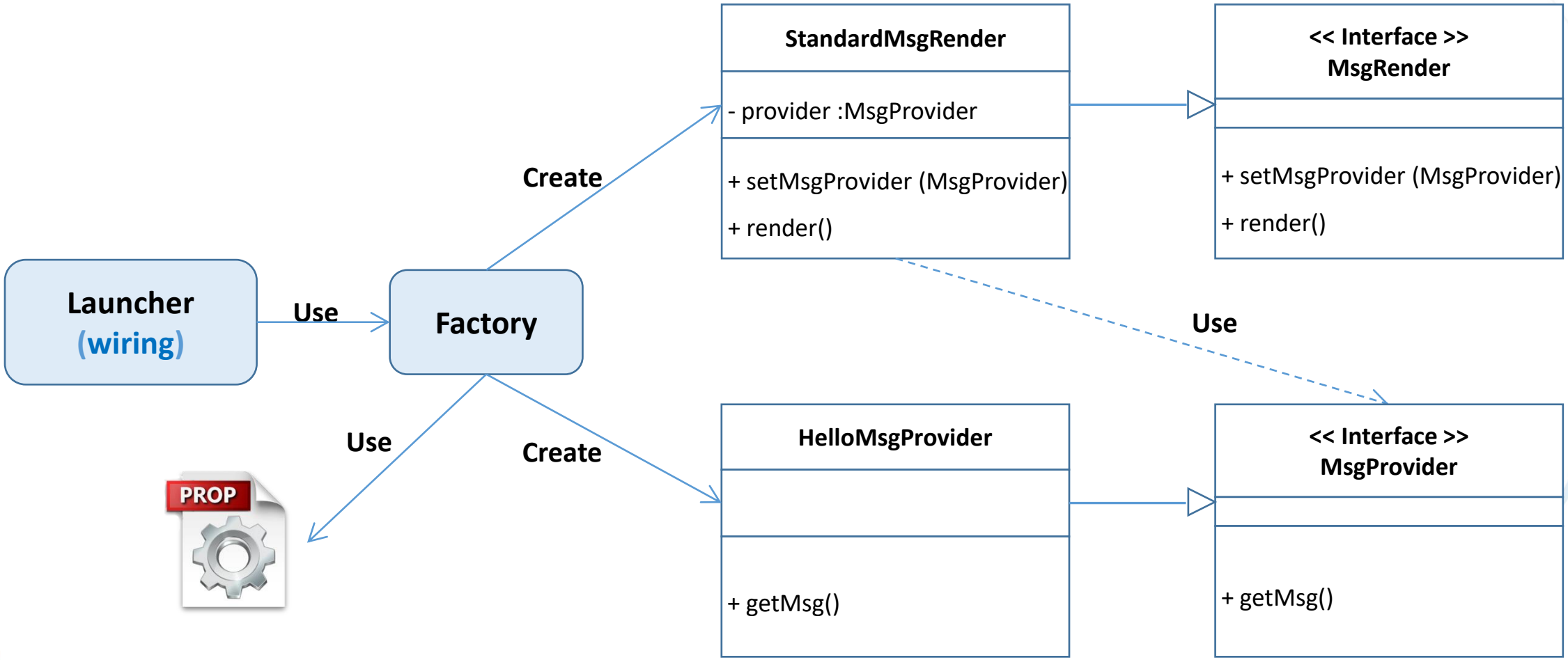
```
public class HelloWorld {

    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.)

- **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.
- **Message Factory:**
  - Which create instance from Message Renderer & Provider and wiring between them.
- **Property File:**
  - Has qualified name of Message Renderer and Message Provider.
- **Launcher:**
  - the main class that use message factory to get instances and wiring them.

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

```
public interface MsgProvider {

    public String getMsg();
}
```

```
public class HelloMsgProvider
        implements MsgProvider {

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

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

```
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.)

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

    public static MessageSupportFactory getInstance() {
        return instance;
    }

    public MsgRender getMsgRender() {
        return renderer;
    }

    public MsgProvider getMsgProvider() {
        return provider;
    }

}
```

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

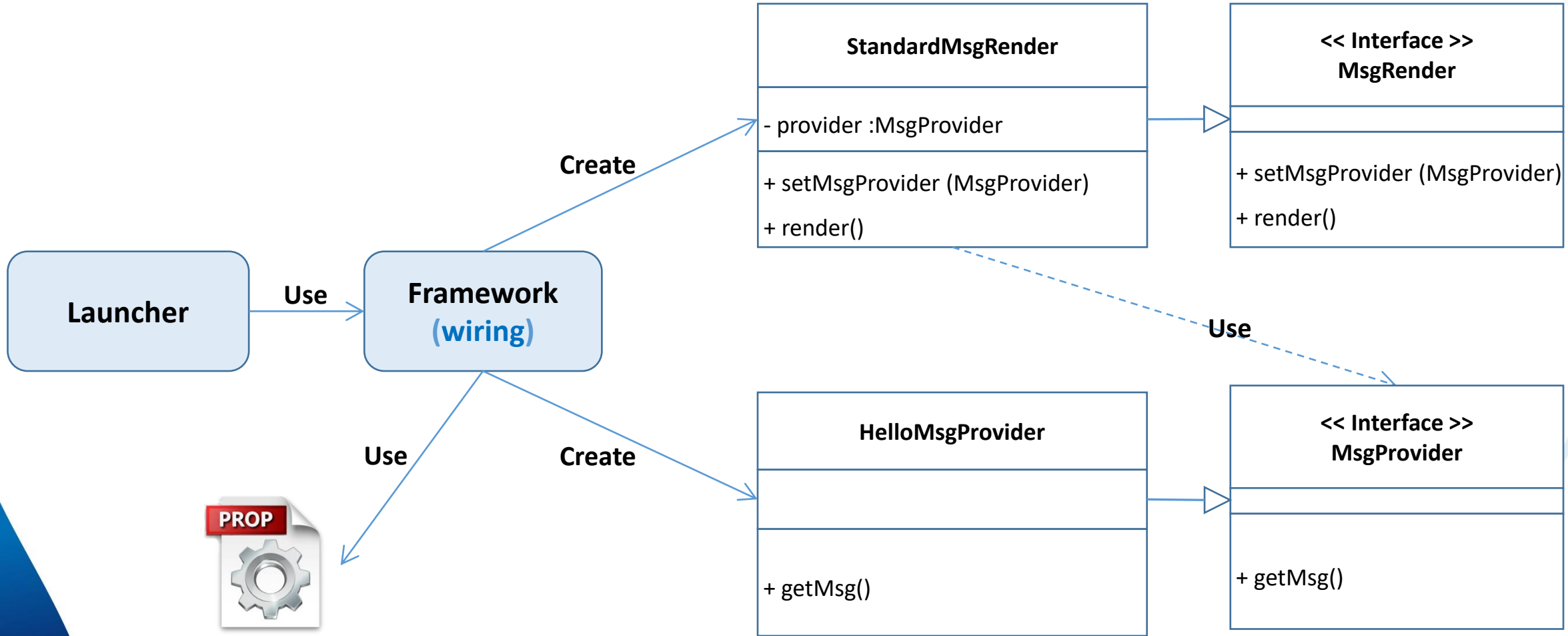
Create Instances from our singleton factory

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.)

- **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.
- **Property File:**
  - Has qualified name of Message Renderer and Message Provider.
- **Launcher:**
  - the main class that use message factory to get instances and wiring them.

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

Java™ Education and Technology Services

```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™ Education and Technology Services

```
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 (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™ Education and Technology Services

```
    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.)

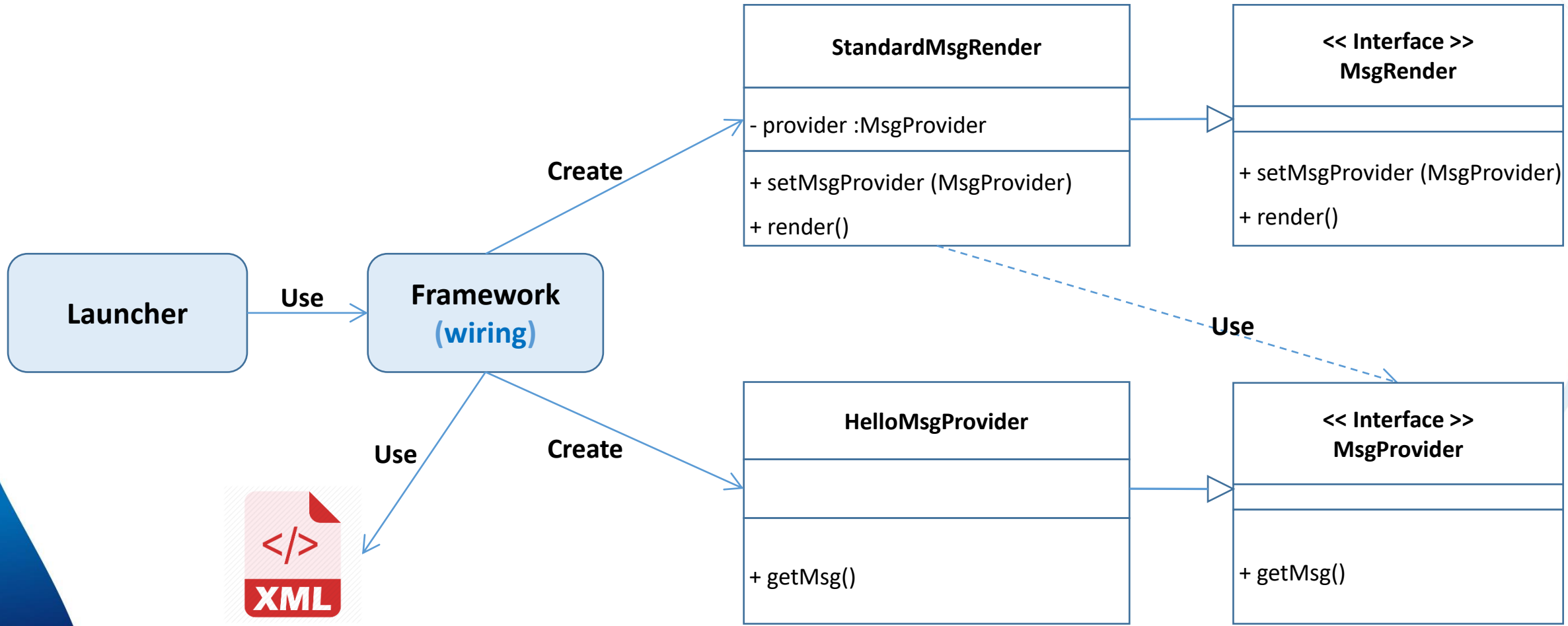
```
public class HelloWorld {

    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.)

- **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.
- **XML File:**
  - Has bean definitions and wiring syntax in structured way.
- **Launcher:**
  - the main class that use message factory to get instances and wiring them.

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

Java™ Education and Technology Services

```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™ Education and Technology Services

```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.)

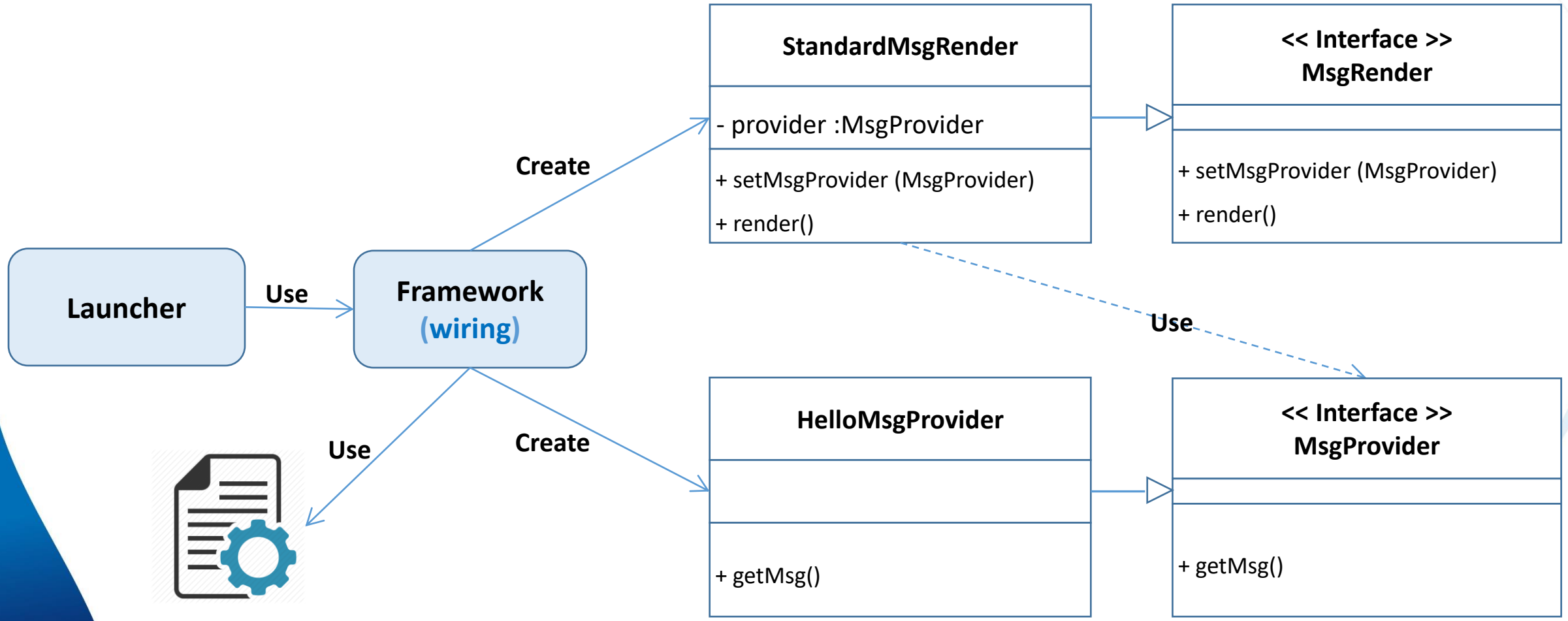

```
public class HelloWorld {

    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™ Education and Technology Services

```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.)

Java™ Education and Technology Services

```java
public interface MsgRender {

    public void setMsgProvider(MsgProvider provider);

    public void render();
}
```

```java
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

```
@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

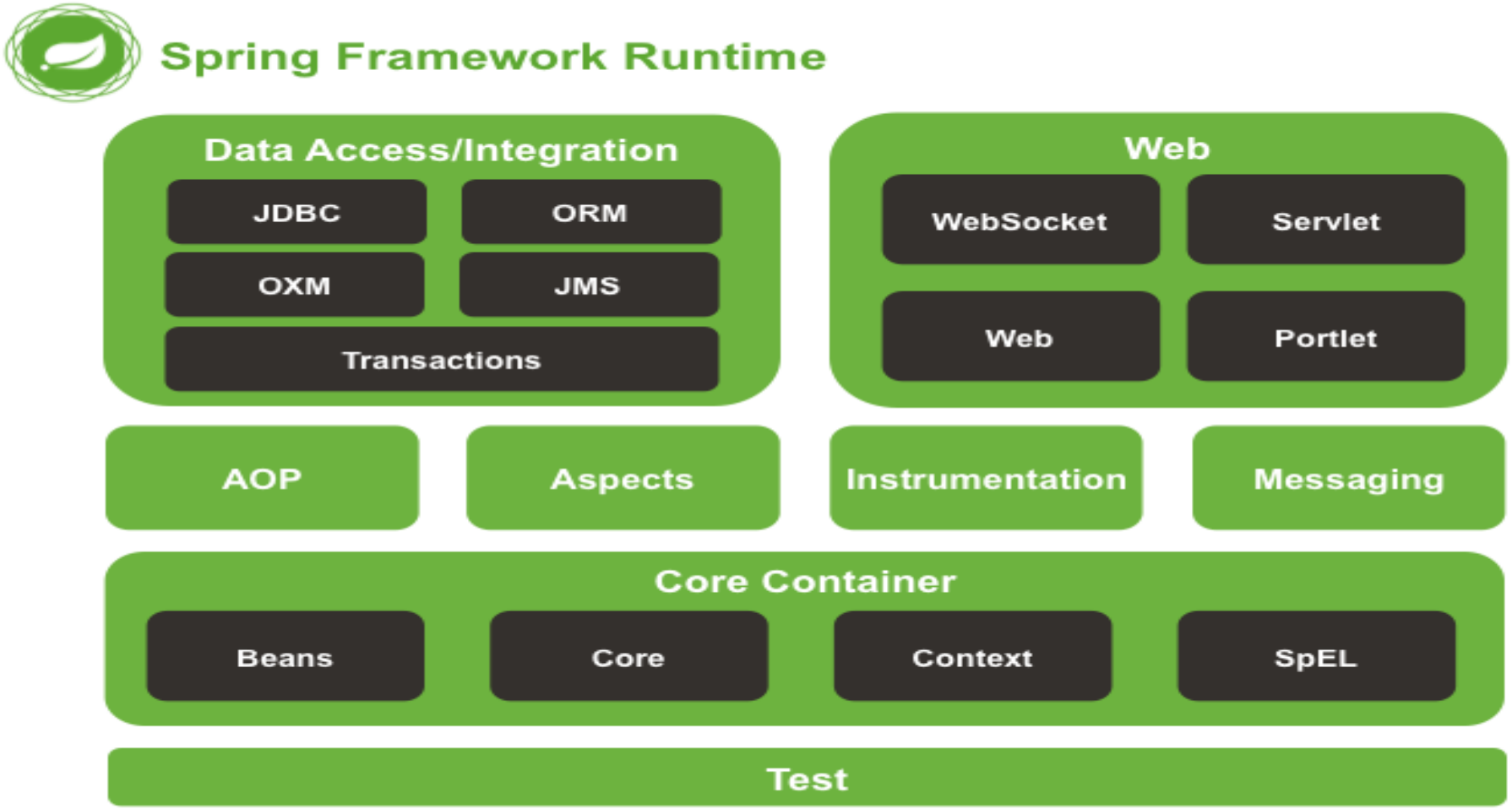
Request an instance from bean named renderer wired with message provider

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:**

```
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 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.)

```
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:**

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

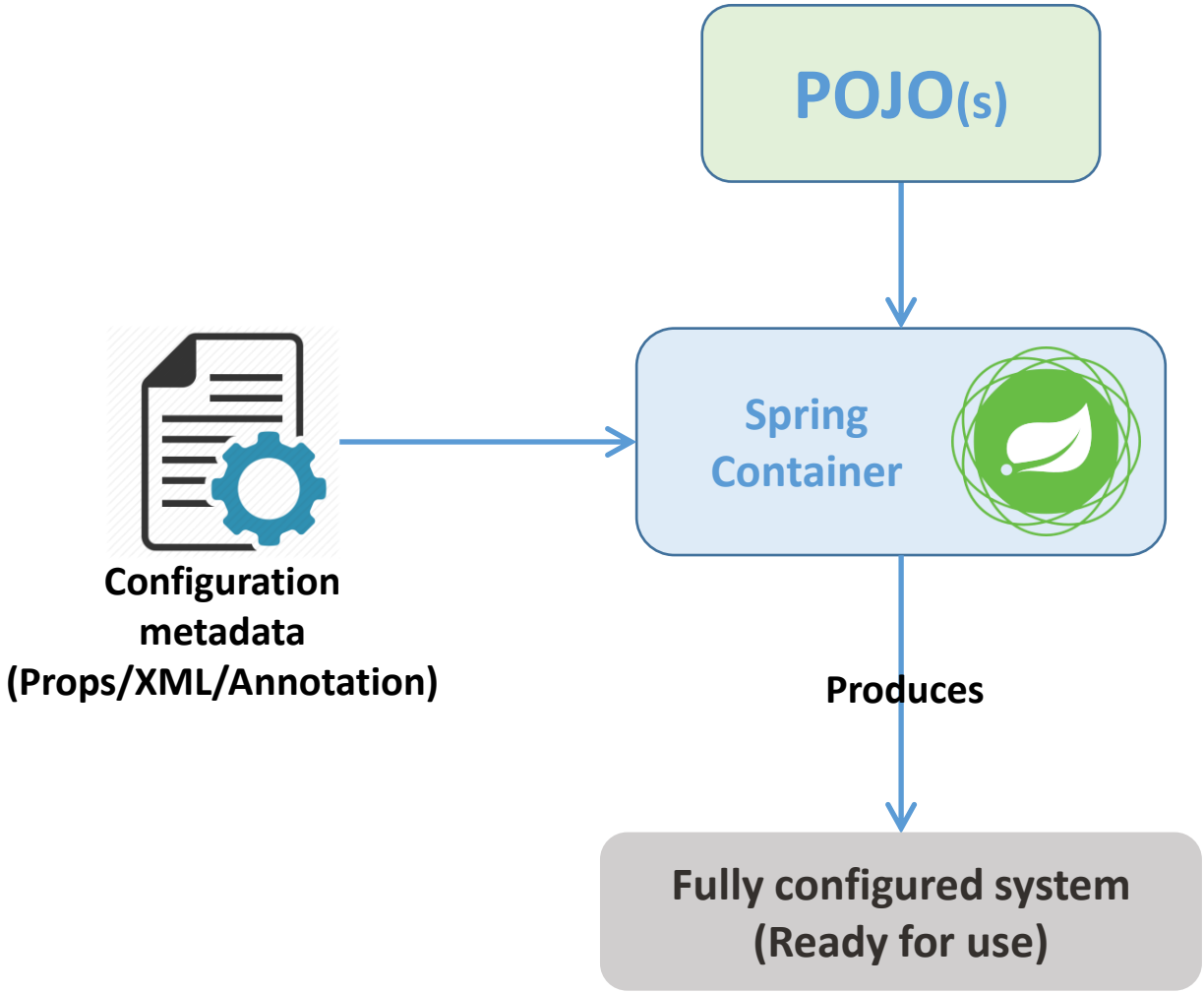
The IoC container (Ex.)

Java™ Education and Technology Services

- **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)

Java™ Education and Technology Services

- 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.)

Java™ Education and Technology Services

- 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/>)

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

- Calling constructor that takes int argument

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

- Calling constructor that takes string argument

```
<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/>)

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

- Calling factory method that takes int argument

```
<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

```
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.

```
public class User {

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

```
<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

```
public class User {

    private String name;
    private float balance;

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

```
<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

```
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".

```
public class DAOServiceImpl implements DAOService {

    private ProductDAO productDAO;

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

Dependency Injection (Ex.)

3. Setter Based Injection

```
<?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 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" .

```
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 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 not recommended.