Spring Framework: Refactoring Helloworld Application (Why DI is useful?)

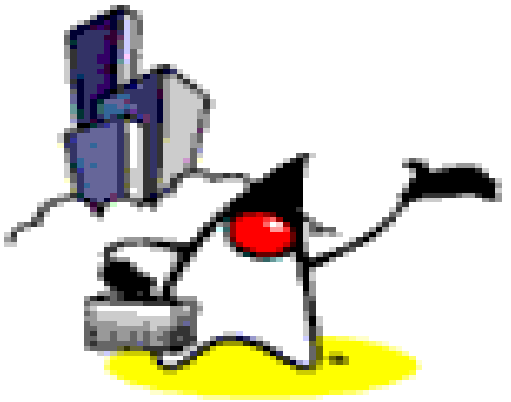
Sang Shin
JPassion.com
"Code with Passion!"

Theme of this Presentation

- How a simple *HelloWorld* application can be refactored in order to achieve the agility (and testability)?
  - How can I change a certain part of an application without affecting other parts of the code? (maintainability)
  - How can I wire different parts of the application without writing a lot of glue code myself? (productivity)
  - How can I test the business logic without being tied up with a particular framework? (testability)
- Dependency Injection (DI) helps in these regards

Refactoring HelloWorld Application

1. HelloWorld
2. HelloWorld with command line arguments
3. HelloWorld with decoupling without using Interface
4. HelloWorld with decoupling using Interface
5. HelloWorld with decoupling through Factory
6. HelloWorld using Spring framework as a factory class but not using DI (Dependency Injection)
7. HelloWorld using Spring framework's DI
8. HelloWorld using Spring framework's DI and XML configuration file
9. HelloWorld using Spring framework's DI and XML configuration file with constructor argument
10. HelloWorld using @Autowired annotation
11. HelloWorld using auto-scanning (component-scanning) in XML
12. HelloWorld using auto-scanning (component-canning) in Java



1. HelloWorld Application

HelloWorld

```
// This is a good old HelloWorld application we all have written
// the first time we learn Java programming.

public class MainApplication {

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

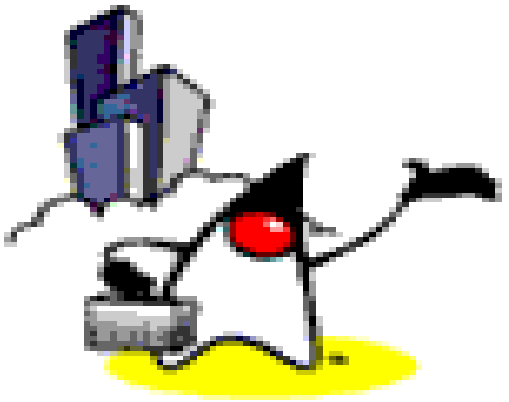
HelloWorld: Outstanding Problems

```
public class MainApplication {

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

- Message content, “Hello World!” in the example above, is hard-coded
- You have to change code (and recompile) to display a different message

HelloWorld: Areas for Refactoring

- Support a flexible mechanism for changing the message



2. HelloWorld Application with Command Line Arguments

HelloWorld With Command Line arguments

```
public class MainApplication {

        public static void main(String[] args) {
            // If an argument is provided, use it, otherwise, display
            // “Hello World!”
            if (args.length > 0) {
                System.out.println(args[0]);
            } else {
                System.out.println("Hello World!");
            }
        }
}
```

HelloWorld With Command Line arguments: Areas Refactored

- This code externalize the message content and read it in at runtime, from the command line argument
  - You can change the message without changing and recompiling the code

HelloWorld With Command Line arguments: Outstanding Problems

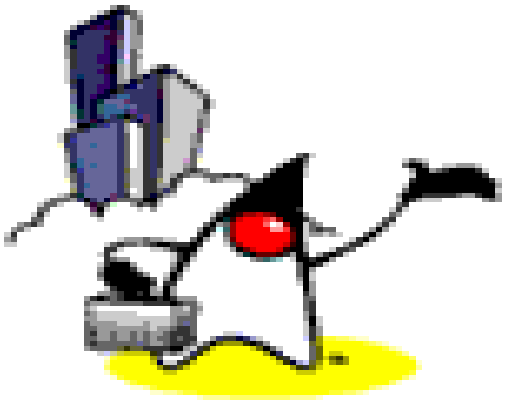
- Rendering logic (renderer) is hard-coded
  - *System.out.println(args[0]);*
  - What if I want to output the message differently, maybe to stderr instead of stdout or send it out to Web service?

- Rendering logic (renderer) is also responsible for obtaining the message (message provider logic)
  - *System.out.println(args[0]);*
  - What if I want to get the message from a Web service instead of from command line argument?
  - Changing how the message is obtained means changing the code in the renderer

HelloWorld With Command Line arguments: Areas for Further Refactoring

- Rendering logic should be in a logically separate code from the rest of the application (encapsulating rendering logic)
  - So that we can change the rendering logic without affecting the rest of the application

- Message provider logic should be in a logically separate code from the rest of the application (encapsulating message provider logic)
  - So that we can change the message provider logic without affecting the rest of the application



3. HelloWorld Application with Decoupling Message Provider & Renderer

Decouple Message Provider

- Encapsulate message provider logic in a separate class - It will decouple message provider logic from the rest of the application

```
public class HelloWorldMessageProvider  {

    // The actual message provider logic can be changed
    // without affecting the rest of the the application
    public String getMessage() {
        return "Hello World!";
    }

}
```

Decouple Message Renderer

- Encapsulate message rendering logic in a separate class - it will decouple message rendering logic from the rest of the application
- Message rendering logic is given *HelloWorldMessageProvider* object by someone

```
public class StandardOutMessageRenderer {

   private HelloWorldMessageProvider messageProvider = null;

   public void setMessageProvider(HelloWorldMessageProvider provider) {
      this.messageProvider = provider;
   }

   // Rendering logic can change without affecting the rest of the application
   public void render() {
      String message  = messageProvider.getMessage();
      System.out.println(message);
   }
}
```

HelloWorld With Decoupling

- Launcher (this is the business logic that creates and uses the message renderer)

```
public class MainApplication {

    public static void main(String[] args) {
        // Create message renderer
        StandardOutMessageRenderer mr =  new StandardOutMessageRenderer();

        // Create message provider
        HelloWorldMessageProvider mp = new HelloWorldMessageProvider();

        // Set the message provider to the message render
        mr.setMessageProvider(mp);

        // Call message renderer
        mr.render();
    }
}
```

Areas Refactored

- Message provider logic and message renderer logic are separated from the rest of the code (they are encapsulated)
  - So message provider (HelloWorldMessageProvider) can change without affecting the rest of the application (StandardOutMessageRenderer, Launcher)
  - So message renderer (StandardOutMessageRenderer) can change without affecting the rest of the application (HelloWorldMessageProvider, Launcher)

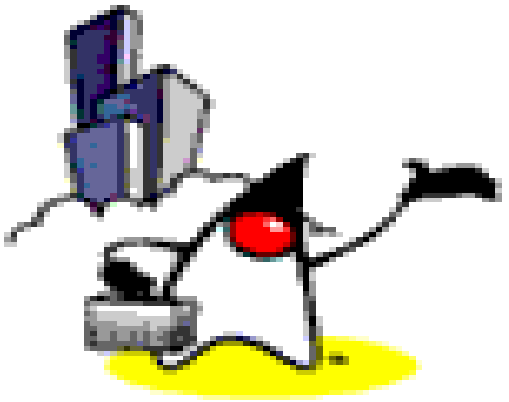
Outstanding Problems

```
public class StandardOutMessageRenderer {

    private HelloWorldMessageProvider messageProvider = null;
    ...
```

- A particular message provider (*HelloWorldMessageProvider)* is hard-coded in the message renderer
  - Usage of a different message provider in the message renderer requires a change in the message renderer
  - What if I want to use *GoodbyeMessageProvider* instead of *HelloWorldMessageProvider*

Areas for Further Refactoring

- Rewrite message renderer and message provider in the form of Java interfaces and their implementations
- User of message renderer and message provider uses interface as a reference type



4. HelloWorld Application with Decoupling Using Interface

HelloWorld With Decoupling (Using Interface)

- Message provider logic now uses Java interface

```
public interface MessageProvider {
    public String getMessage();
}

public class HelloWorldMessageProvider
                        implements MessageProvider {

    public String getMessage() {
        return "Hello World!";
    }

}
```

MessageRenderer and MessageProvider Interfaces

- Message rendering logic is given *MessageProvider* object instance by someone

```
public interface MessageRenderer {

    public void render();
    public void setMessageProvider(MessageProvider provider);
}
```

StandardOutMessageRenderer Class

```
public class StandardOutMessageRenderer
                        implements MessageRenderer {

    // MessageProvider is Java Interface
    private MessageProvider messageProvider = null;

    public void render() {
        System.out.println(messageProvider.getMessage());
    }

    // MessageProvider is Java Interface
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }
}
```

HelloWorld With Decoupling (using Interface)

- Launcher

```
public class MainApplication {

    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

Areas Refactored

- Message rendering logic does not get affected by the change in message provider implementation because it uses the *MessageProvider* interface as a reference type

Outstanding Problems

```
public class MainApplication {

    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

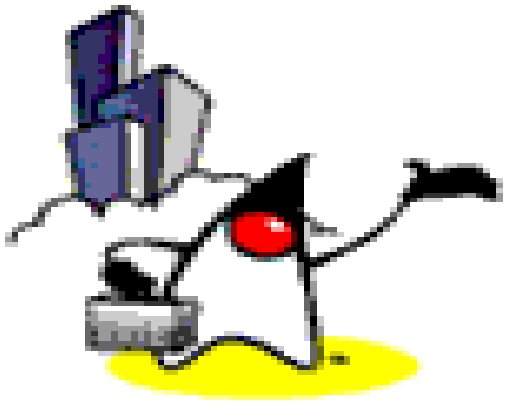
- The implementations of renderer and message providers are hard-coded in the launcher
- Using different implementation of either means a change to the launcher

Areas for Further Refactoring

- Create a simple factory class that reads the implementation class names from a properties file and instantiate them during runtime on behalf of the application

Lab:

Exercise 1 - 4
4933_spring_helloworld.zip



5. HelloWorld Application with Decoupling through Factory class

Factory Class That Read Properties File

```
public class MessageSupportFactory {
    private static MessageSupportFactory instance = null;
    private Properties props = null;
    private MessageRenderer renderer = null;
    private MessageProvider provider = null;

    private MessageSupportFactory() {
        props = new Properties();
        try {
            props.load(new FileInputStream("application.properties"));

            // Get the name of the implementation classes
            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");

            // Create object instances of MessageRenderer and MessageProvider
            renderer = (MessageRenderer) Class.forName(rendererClass).newInstance();
            provider = (MessageProvider) Class.forName(providerClass).newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
    // continued to the next slide
```

HelloWorld With Factory Class

```
    // continued from the previous slide
    static {
        instance = new MessageSupportFactory();
    }

    public static MessageSupportFactory getInstance() {
        return instance;
    }

    public MessageRenderer getMessageRenderer() {
        return renderer;
    }

    public MessageProvider getMessageProvider() {
        return provider;
    }

}
```

Launcher Code (Business Logic)

```
public class HelloWorldDecoupledWithFactory {

    public static void main(String[] args) {
        MessageRenderer mr =
            MessageSupportFactory.getInstance().getMessageRenderer();
        MessageProvider mp =
            MessageSupportFactory.getInstance().getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

HelloWorld With Factory Class: Properties file

```
renderer.class=StandardOutMessageRenderer
provider.class=HelloWorldMessageProvider
```

Areas Refactored

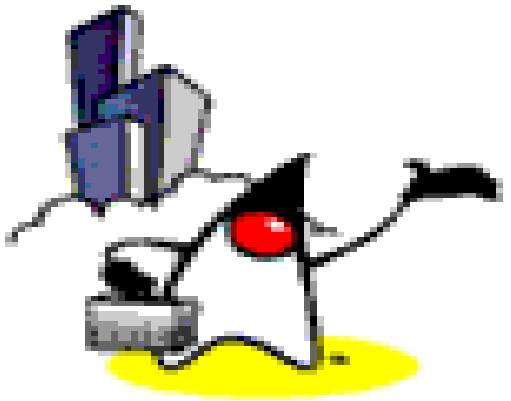
- Message provider implementation and Message renderer implementation can be replaced simply by changing the properties file
- No change is required in the launcher code even when different message provider implementation and message renderer implementation need to be used

Outstanding Problems

- You still have to write a lot of glue code yourself to assemble the application together
  - You have to write *MessageSupportFactory* class

Areas for Further Refactoring

- Replace *MessageSupportFactory* class with Spring framework's *DefaultListableBeanFactory* class
  - You can think of *DefaultListableBeanFactory* class as a more generic version of *MessageSupportFactory* class



6. HelloWorld Application with Spring Framework (but not using DI yet)

HelloWorld With Spring Framework

```
public class MainApplication {

    public static void main(String[] args) throws Exception {

        // Get the bean factory - the code of getBeanFactory() is in the next slide
        BeanFactory factory = getSpringBeanFactory();

        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        MessageProvider mp = (MessageProvider) factory.getBean("provider");

        mr.setMessageProvider(mp);
        mr.render();
    }
```

Areas Refactored

- Removed the need of your own glue code (*MessageSupportFactory*) instead use Spring's built-in factory class
- Gained a much more robust factory implementation with better error handling and fully de-coupled configuration mechanism

Outstanding Problems

- Spring acts as no more than a sophisticated factory class creating and supplying instances of classes as needed in this case

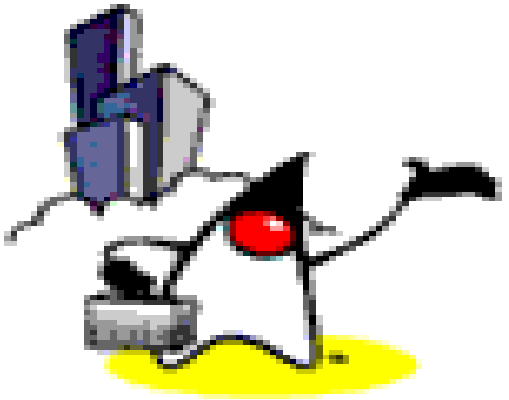
- The launcher code must have knowledge of the MessageRenderer's dependencies and must obtain dependencies and pass them to the MessageRenderer (This is called “wiring”)
  - You still have to inject an instance of *MessageProvider* into the implementation of *MessageRenderer* yourself

Areas for Further Refactoring

- Use Dependency Injection (DI) of the Spring Framework
  - Let Spring framework to handle the wiring – Let Spring framework inject MessageProvider object into MessageRenderer object
  - You give this wiring instruction through a Spring configuration

Lab:

Exercise 5 - 6
4933_spring_helloworld.zip



8. HelloWorld Application with Spring Framework & Dependency Injection (DI) using XML Configuration File

Spring DI with XML file

- Dependencies of beans are specified in an XML file
  - XML based bean configuration is more popular than properties file based configuration

Spring DI with XML Configuration File

```
<beans>
    <bean id="renderer"
          class="StandardOutMessageRenderer">
        <property name="messageProvider"
                  ref=”provider"/>
    </bean>
    <bean id="provider"
          class="HelloWorldMessageProvider"/>
</beans>
```

Spring DI with XML Configuration File

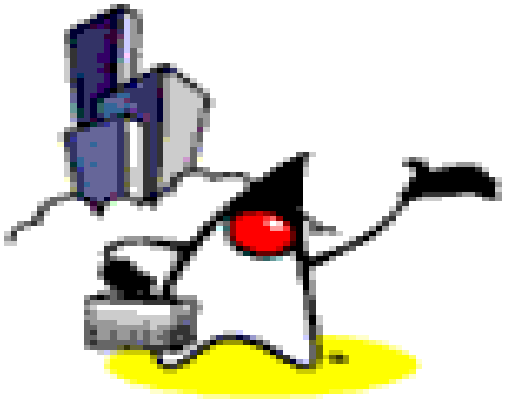
```
public class MainApplication {

    public static void main(String[] args) throws Exception {

        // get the bean factory
        BeanFactory factory = getSpringBeanFactory();
        MessageRenderer mr =
               (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getSpringBeanFactory() throws Exception {
        // get the bean factory
        BeanFactory factory =
             new ClassPathXmlApplicationContext("beans.xml");

        return factory;
    }
}
```

Areas Refactored

- The main() method now just obtains the *MessageRenderer* bean and calls *render()*
  - It does not have to obtain MessageProvider bean and set the MessageProvider property of the MessageRenderer bean itself.
  - This “wiring” is performed through Spring framework's Dependency Injection.

A Few Things to Observe

- Note that we did not have to make any changes to the classes (beans) that are being wired together
- These classes have no reference to Spring framework whatsoever and completely oblivious to Spring framework's existence
  - No need to implement Spring framework's interfaces
  - No need to extend Spring framework's classes
- These classes are genuine POJO's which can be tested without dependent on Spring framework



9. HelloWorld Application with Spring Framework & Dependency Injection (DI) using XML Configuration File with Constructor argument

Spring DI with XML Configuration File: via Constructor

```
<beans >
 <bean id="renderer"
  class="com.javapassion.examples.StandardOutMessageRenderer">
     <constructor-arg ref="provider"/>
 </bean>
 <bean id="provider"
  class="com.javapassion.examples.HelloWorldMessageProvider">
 </bean>
</beans>
```

Spring DI with XML Configuration File: via Constructor

```
public class StandardOutMessageRenderer
    implements MessageRenderer {

    private MessageProvider messageProvider = null;

    public StandardOutMessageRenderer(MessageProvider provider){
      this.messageProvider = provider;
    }

    public void render() {
        //...

        System.out.println(messageProvider.getMessage());
    }

}
```

Areas Refactored

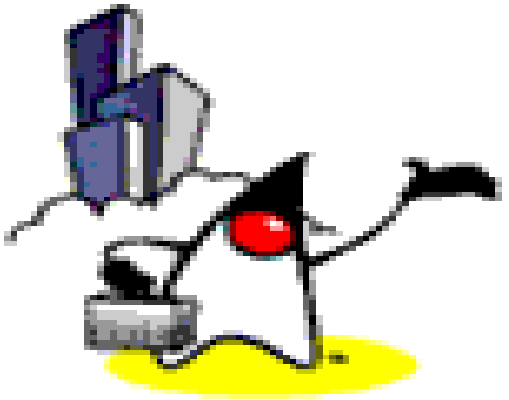
- We were able to specify wiring requirements in XML
- What if there are too many wiring requirements that need to be specified in XML?
  - Declaring wiring requirements among beans could be unmanage'able if there are too many

Areas for Further Refactoring

- We want to specify wiring requirements in the source code using Java Annotation

Lab:

Exercise 8-9
4933_spring_helloworld.zip



10. HelloWorld Application with Spring Framework & Dependency Injection (DI) using @Autowired Annotation

Wiring through @Autowired Annotation in Code

```
public class StandardOutMessageRenderer implements MessageRenderer {

    @Autowired
    private MessageProvider messageProvider = null;

    public void render() {
        // ...
    }

    // Not needed anymore since it will be autowired
    // public void setMessageProvider(MessageProvider provider) {
    //    this.messageProvider = provider;
    // }

}
```

No need to specify wiring requirements in the XML

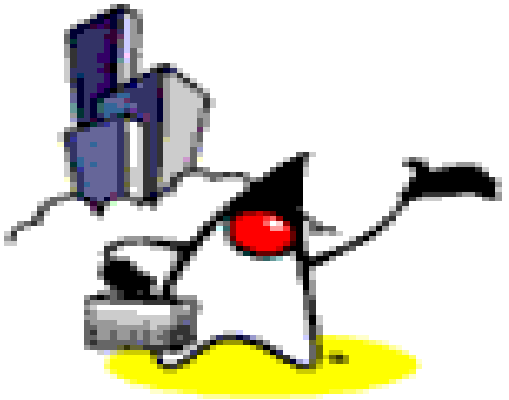
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans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

    <!-- enable the usage of annotations -->
    <context:annotation-config />

     <!-- Note that there is no messageProvider property in the renderer.
          It is because the wiring requirement is specified through
          @Autowired annotation  -->
    <bean id="renderer"
        class="com.javapassion.examples.StandardOutMessageRenderer">
    </bean>

    <bean id="provider"
        class="com.javapassion.examples.HelloWorldMessageProvider"/>
</beans>
```

Areas Refactored

- We were able to specify wiring requirements right in the source code using @Autowired annotation
- But we still have to declare all the beans in the XML file
- What if there are too many beans we need to declare
  - Having to declare all the beans in the XML is a chore
  - Can we have Spring framework to auto-scan and detect all the beans?

Areas for Further Refactoring

- We want Spring framework to auto-detect all the beans and create instances



11. HelloWorld Application with Spring Framework & Dependency Injection (DI) using Auto-scanning

Auto-scanning beans

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

    <!-- enable the usage of annotations -->
    <!--  <context:annotation-config /> -->

    <!-- scan component, it also assumes annotation-config as well-->
    <context:component-scan base-package="com.javapassion.examples"/>
</beans>
```

Bean with @Component Annotation gets detected

```
@Component("renderer")   // This is the same as @Component(value="renderer")
public class StandardOutMessageRenderer implements MessageRenderer {

    @Autowired
    private MessageProvider messageProvider = null;

    public void render() {
        // ...
    }

    // Not needed anymore since it will be autowired
//    public void setMessageProvider(MessageProvider provider) {
//        this.messageProvider = provider;
//    }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

Bean with @Component Annotation gets detected

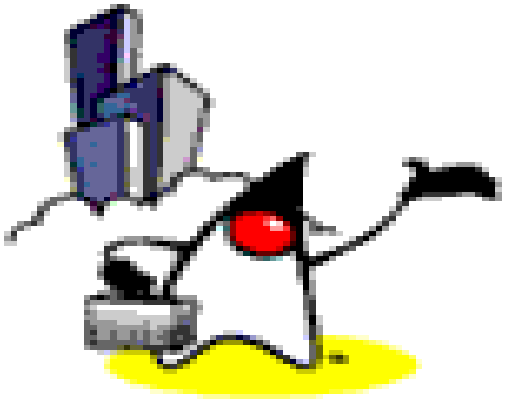
```
@Component
public class HelloWorldMessageProvider implements MessageProvider {

    public String getMessage() {

        return "Hello World!";
    }

}
```

Areas Refactored

- We don't have to declare beans in the XML file anymore since the beans are auto-scanned

Areas for Further Refactoring

- We still have any XML file, which indicates to the Spring framework auto-scanning is required
- We want to get away from XML all together - we can use @Configuration and @ComponentScan annotations



11. Use @Configuration & @ComponentScan

@Configuration & @ComponentScan

```
@Configuration
@ComponentScan
public class MainApplication {

   public static void main(String[] args) throws Exception {

      ApplicationContext context
            =  SpringApplication.run(MainApplication.class,args);
      MessageRenderer mr = context.getBean(MessageRenderer.class);
      mr.render();
   }

}
```

Areas Refactored

- Use Java configuration using @Configuration - No more beans.xml
- Component scanning is done through @ComponentScan

Lab:

Exercise 10-12
4933_spring_helloworld.zip

Code with Passion!
JPassion.com