selsoft
build better, deliver faster

Enterprise Application Development with Spring

Chapter 6: Annotation-Based Configuration

Instructor
Akın Kaldıroğlu
Expert for Agile Software Development and Java

Topics

- **Annotation-Based Configuration**
  - @Autowired
- **Defining Beans Outside of XML**
  - @Component and Component Scan
  - Qualifying Beans: @Primary, @Qualifier & Custom Qualifiers
  - @Scope
  - Value Injection using @Value
  - Life-Cycle Events
  - @Order & @DependsOn
- **XML vs. Annotation Configuration**

Annotation-Based Configuration

selsoft

build better, deliver faster

Anotation-Based Configuration - I

- Configuring all of the beans and their dependencies using constructor and property arguments in XML is cumbersome and error-prone.
- So **Spring** developed along the way easier ways to do the same thing.
- **Spring** provides annotations for the specification of both
  - beans and
  - dependencies.
- We'll start with dependency configuration through annotations and later proceed to bean annotations.

Autowiring - I

- **Spring** container can figure out and satisfy dependencies between beans using annotations since version 2.5.
- **Spring** can resolve collaborators automatically by inspecting the contents of the `ApplicationContext`.
- This is called **autowiring**.
- Autowiring allows cleaner DI management.

Autowiring - II

- To have annotation-based configuration **`<context:annotation-config/>`** element in **`<beans/>`** element must exist in XML file.
- Context schema is available at http://www.springframework.org/schema/context/spring-context-4.3.xsd
- Eventually **Spring** will let us to get rid of all XML files and depend on annotations only.

Packages and Annotations

- Spring has its annotations for DI mainly in following two packages:
  - `org.springframework.beans.factory.annotation` has `Autowired`, `Configurable`, `Lookup`, `Qualifier`, `Required`, and `Value`.
  - `org.springframework.context.annotation` has mainly `Bean`, `ComponentScan`, `Conditional`, `Configuration`, `DependsOn`, `Import`, `Lazy`, `Primary`, `PropertySource`(`s`), and `Scope`.

@Autowired

@Autowired - I

- `org.springframework.beans.factory.annotation.Autowired` annotation is used to specify dependencies in Java source code.
    - `Autowired` annotation became available with **Spring** 2.5.
- `<context:annotation-config/>` element in `<beans/>` is needed.
- **Spring** figures out dependencies through `Autowired` annotations.
- But beans must still be defined in XML configuration without any info on their dependencies.

@Autowired - II

- `Autowired` annotation is applied to following places and put before:
  - instance variable
  - constructor
  - setter method
  - any configuration method with any number of parameters
- Of course any element that is annotated by `Autowired` should be injectable with a bean defined in `<bean/>` in the XML file.

@Autowired - III

- `Autowired` annotation has only one attribute of type `boolean`, `required` which is `true` in default.
- So **Spring** tries to inject dependencies into every single point that is annotated by `Autowired`.
  - If it can not satisfy any required dependency it throws `org.springframework.beans.factory.UnsatisfiedDependencyException`.
- Giving `required` attribute `false` makes the dependency optional which may cause `NullPointerException`.

@Autowired - IV

- In a class only one constructor can be annotated with `Autowired`.
- A class can have many other constructors without annotation but Spring always try to call the one with annotation to satisfy all dependencies.
- If there is only one constructor in a bean with an injectable dependency there is no need even to use `@Autowired` for that constructor.
  - **Spring** automatically wires the dependencies by passing it to the constructor of the bean.
    - This feature became available with 4.3.

@Autowired - V

- If the dependency is for a value instead of a bean then values must be specified for injection in XML file using **`value`** attribute of **`</bean>`**.

- In this case **`constructor-arg`** and **`property`** attributes are used only for values, there is still no need to use **`ref`** for beans because they are defined in XML and **Spring** automatically finds and autowires them.

```java
public class BeanC {
  ..
  @Autowired
  public BeanC(String nameOfBean, BeanD beanD){
     this.nameOfBean = nameOfBean;
     this.beanD = beanD;
  }
  ..
}
```

```xml
<beans>
 …
  <bean id="beanC"
          class="org.javaturk.spring.di.ch06.autowired.domain.BeanC">
          <constructor-arg name="nameOfBean" value="BEAN-C"  />
  </bean>

  <bean id="beanD"
          class="org.javaturk.spring.di.ch06.autowired.domain.BeanD"/>
 …
</beans>
```

@Autowired - VI

- **Spring** injects into any method that accepts a parameter of type of the collaborator bean.
- Spring calls them **config methods**.
- Property setter methods are special case of config methods due to their proper names.
- **Spring** does not put forward any other rule on the config methods.
    - Config methods can return values for example.

@Autowired - VII

- Beans can be excluded from autowiring by setting the **`autowire-candidate`** attribute of the **`<bean/>`** element to **`false`**.
- The IoC container makes that specific bean definition unavailable to the autowiring infrastructure including annotation style configurations such as **`@Autowired`**.

@Autowired - VIII

- If **`@Autowired`**. is used on a method that doesn't receive any dependency IoC raises a log at run-time and says *INFO: Autowired annotation should only be used on methods with parameters:…*

AutowiredExample

- `org.javaturk.spring.di.ch06.autowired.AutowiredExample`
  - Notice how injections to constructors and setters are made.
  - Some of dependencies are for beans and some others for values.
    - Notice how they are specified in XML file.

greeting11

- `org.javaturk.spring.di.ch06.greeting.greeting11.Application`
  - `getBean1()`: Put `Autowired` annotation to four different places in `StandardOutputRenderer` and observe how the dependency is injected.
  - `getBean1()`: Remove `Autowired` annotations to observe observe how the dependency is injected into the unique constructor.
    - Run it with one and two constructors.

greeting11

- **`org.javaturk.spring.di.ch06.greeting.greeting11.Application`**
  - **`getBean1()`**: Use **`Autowired(required=false)`** to observe optional dependency.
    - Observe that it may lead to **`NullPointerException`**.
  - **`getBean2()`**: More than one bean can be injected into the same method annotationed by **`Autowired`**.

selsoft
build better, deliver faster

@Required

@Required - I

- `org.springframework.beans.factory.annotation.Required` is the original annotation to specify that the dependency is required.
- `@Required` can only be used with setter methods.
- `@Required` only marks the setter method that the dependency is required and must to be satisfied using either `property` attribute of `</bean>` in XML file or `@Autowired` otherwise `org.springframework.beans.factory.BeanInitializationException` with a message that *the property is required* is thrown.

@Required - II

- **`@Required`** avoids having **`NullPointerException`**.
- **`org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor`** enforces required JavaBean properties to have been configured.
- **`@Required`** and **`RequiredAnnotationBeanPostProcessor`** have been deprecated in version 5.1 so use **`Autowired`** instead.
- There is no need to use them anymore.

Application

- `org.javaturk.spring.di.ch06.required.Application`
    - First run it without any `@Required` annotation to get `null` references or `NullPointerException`.
    - Then run it with `@Required` annotation to satisfy setter dependencies.
    - Observe what would happen if the `@Required` dependency is not satisfied.

Defining Beans Outside of XML

selsoft

build better, deliver faster

Defining Beans Outside XML

- Using `<context:annotation-config/>` element in `<beans/>` and `Autowired` annotation allows **Spring** to figure out only dependencies but beans must still be defined in XML files.
- **Spring** allows to specify beans outside the XML configuration file using two mechanisms:
  - `Component` annotation
  - `Bean` factory methods
    - In these mechanisms beans are defined in Java source code using specific annotations.

@Component and Component Scan

Copmponent - I

- `org.springframework.stereotype.Component` is an annotation that indicates that the annotated class is a **component**.
- `Component` is placed before class declaration and makes its instances beans.
- Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
- Classpath scanning for components is possible by `<context:component-scan/>` in `<beans/>` in XML file.

Scanning Components - I

- `<context:component-scan/>` element has an attribute called **`base-package`** to specify where to start scanning beans.
- Using **`base-package`** is mandatory and it receives a comma/ semicolon/space/tab/linefeed-separated list of packages to scan for annotated components.
- **Spring** scans all of the packages and their sub-packages listed in **`base-package`** for annotated components that will be auto-registered as beans.

Scanning Components - II

- Component scan and use of `Component` annotation make XML configuration files free of any bean definitions and keem them very short.
- `<context:component-scan/>` implicitly enables the functionality of `<context:annotation-config>` which enables the use of `Autowired` annotation.
    - So no need to include the `<context:annotation-config>` element when `<context:component-scan/>` is used.

Component - II

- `Component` has only one attribute called `value` and it takes a `String` for the name of the bean.
- If called `value` is not used **Spring** uses as default name the simple name of the class of the bean with its first letter converted to lower camel case.

```
@Component // name is standardOutputRenderer
public class StandardOutputRenderer {  …}
```

```
@Component(“renderer”)
public class StandardOutputRenderer {…}
```

```
@Component(value=“renderer”)
public class StandardOutputRenderer {…}
```

Component - IV

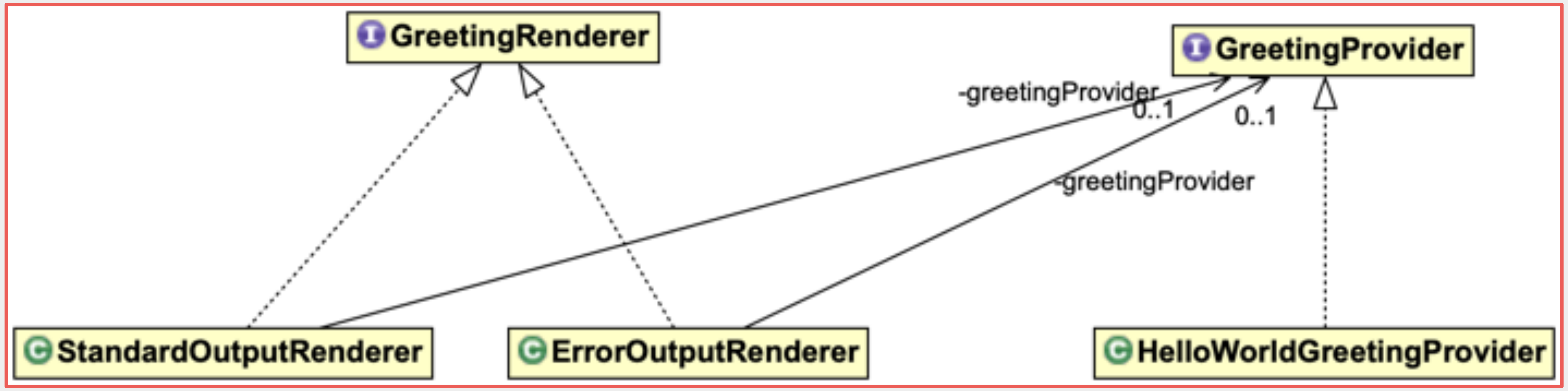
- `Component` must be used for classes so that their objects can be created.
- So using `Component` for interfaces or abstract classes does not make sense.

Component - V

- In fact **Spring** provides many different components for different tasks:
  - `Component`, `Repository`, `Service`, `Controller`, `RestController`, `ControllerAdvice`, and `Configuration`
- These stereotypes are all annotations for **Spring** beans and they are automatically detected using classpath scanning.
- We will discuss them in future.

greeting12

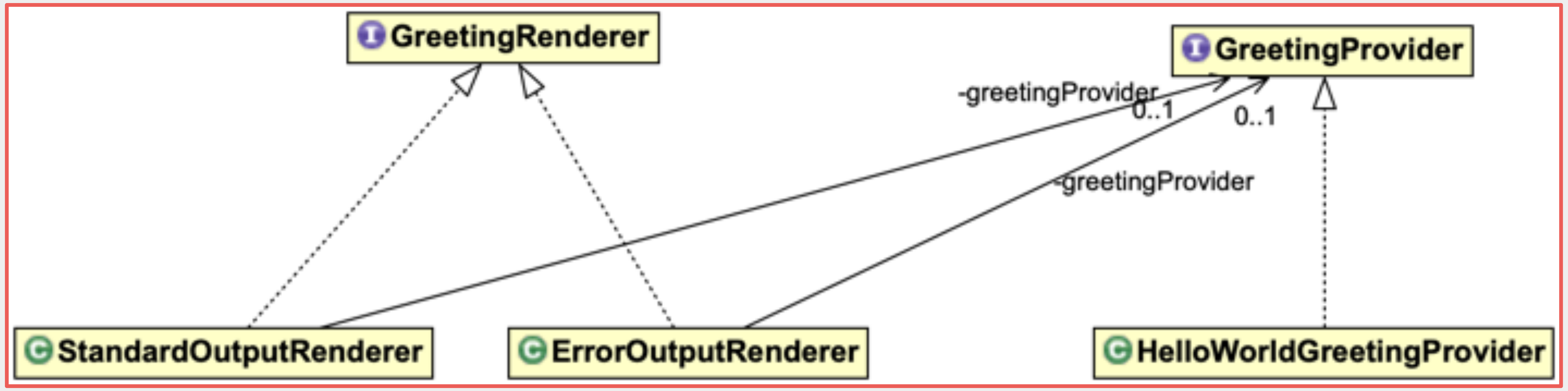
- `org.javaturk.spring.di.ch06.greeting.greeting12.Application`
  - Observe the default names for the components.
  - Observe the effect of `base-package` in `<context:component-scan/>`.



Qualifying Beans

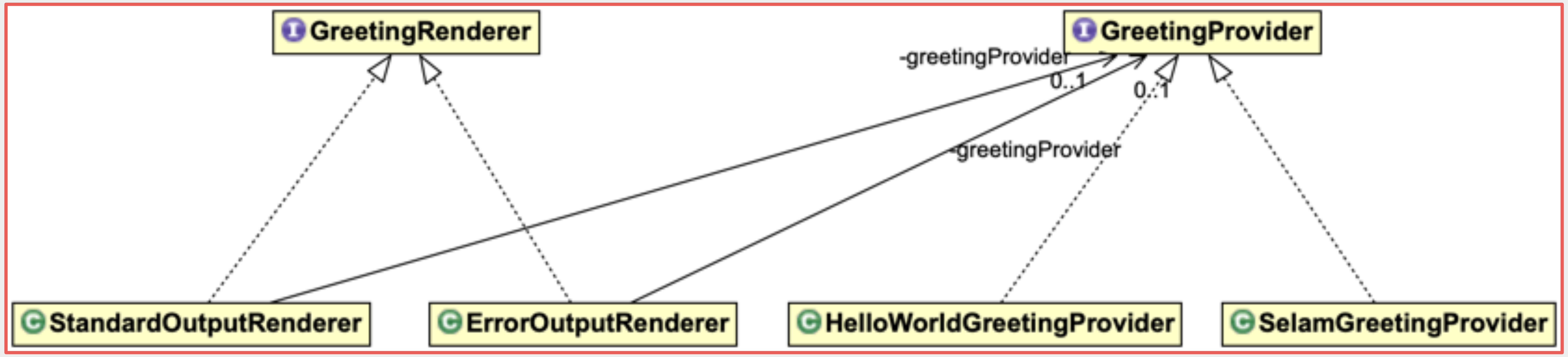
How To Select Among Beans? - I

- Think about the model used in `greeting12`:
  - `GreetingRenderer` implementations are injected `GreetingProvider` using `@Autowired`.
  - What would happen if there were more than one implementation of `GreetingProvider`?



greeting13

- **`org.javaturk.spring.di.ch06.greeting.greeting13.Application`**
  - Observe that **`GreetingProvider`** has two implementations both of which are annotated by **`@Component`**.



How To Select Among Beans? - II

- Spring throws **`UnsatisfiedDependencyException`** when it gets confused regarding which bean to inject.
  - **`NoUniqueBeanDefinitionException`** is the nested exception with the message like *No qualifying bean of type … vailable: expected single matching bean but found 2:*

How To Select Among Beans? - II

- There are several solutions for the problem:
  - Naming convention: Using matching names for both bean and variable or parameter name at the injection point.
  - Using `Primary` and `Qualifier` annotations.
  - Qualification through generics.

Using Matching Names - I

- If the names of the component and the variable or parameter at injection point are the same then **Spring** uses the bean with matching name.
- If the names of the candidate beans for the injection clash then the same problem occurs.

```
@Component // name is helloWorldGreetingProvider
public class HelloWorldGreetingProvider implements GreetingProvider{
  …
}
```

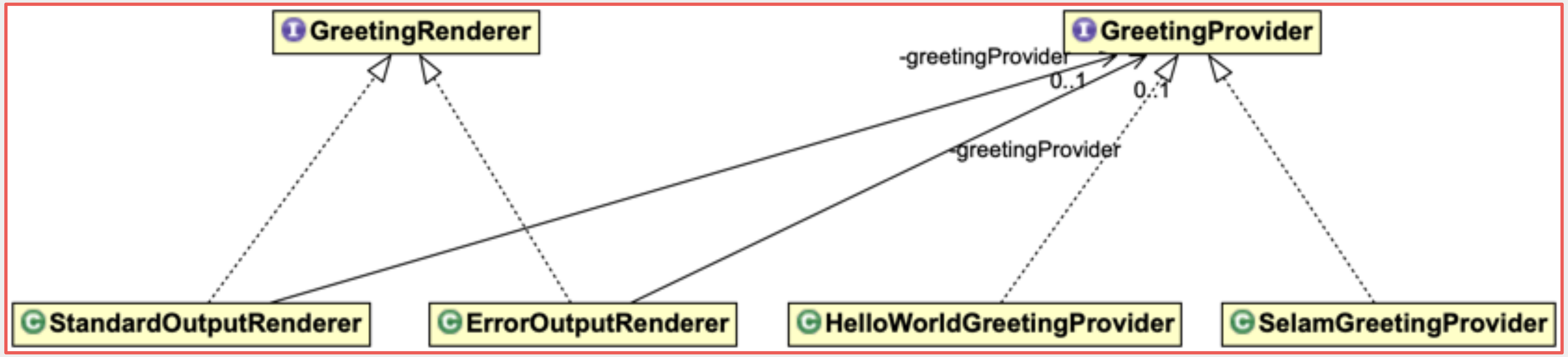
```
@Component
public class StandardOutputRenderer {
  @Autowired
  private GreetingProvider helloWorldGreetingProvider
  …
}
```

Using Matching Names - II

- This solution requires using component names in their clients.
- And it relies on naming convention.

greeting13

- **`org.javaturk.spring.di.ch06.greeting.greeting13.Application`**
  - **`GreetingProvider`** has two implementations both of which are annotated by **`@Component`**.
  - Observe how the injecton is resolved through naming convention.



Qualifying Beans

@Primary

@Primary - I

- `org.springframework.context.annotation.Primary` is an annotation to give the annotated bean preference when there are more than one bean available as candidate for injection.
- And off course if there are more than one bean with `@Primary` annotation `Spring` throws `UnsatisfiedDependencyException` with a nested exception `NoUniqueBeanDefinitionException` saying that `more than one 'primary' bean found among candidates`.

@Primary - II

- Only one bean can be made primary and to select other beans qualifiers should be used.
  - That’s because using `@Primary` invalidates qualification through naming.
- And for the same purpose `<bean>` element's `primary` attribute can be used in XML.
  - In which case beans must be defined in the XML.

greeting14

- **`org.javaturk.spring.di.ch06.greeting.greeting14.Application`**
  - **`getBeans1()`**
  - Remove all **`@Qualifed`** annotations and **`</qualifier>`** elements in XML file and use **`@Primary`** for a bean instead.
  - Observe the same effect with **`primary`** attribute of **`<bean>`** element in XML.

selsoft
build better, deliver faster

Qualifying Beans

@Qualifier

Qualifier - I

- `org.springframework.beans.factory.annotation.Qualifier` offers a better solution for this problem.
- It is used on a field or parameter as a qualifier for candidate beans when autowiring.
- `Qualifier` annotation takes only one `String` argument as `value` which serves as a qualifier to differentiate among the candidate beans.
    - `value` should typically take the name of the one of the candidate components for the injection.

Qualifier - II

- `Qualifier` should be used where `@Autowired` exists.
- If `Qualifier` uses the name of the target bean intended for injection than the injection happens.

```
@Component
public class StandardOutputRenderer {
  @Autowired
  @Qualifier("helloWorldGreetingProvider")
  private GreetingProvider greetingProvider

 @Autowired
  public void setGreetingProvider(@Qualifier("hello") GreetingProvider greetingProvider){
     this.greetingProvider = greetingProvider;
  }
…
}
```

```
@Component // name is helloWorldGreetingProvider
public class HelloWorldGreetingProvider
            implements GreetingProvider{
  …
}
```

Qualifier - III

- It is possible to use **`@Qualifier`** with **`@Component`** too.
- In this case the values of both **`@Qualifier`** annotations must match.

```
@Component
public class StandardOutputRenderer {
  @Autowired
  @Qualifier(“hello”)
  private GreetingProvider greetingProvider

 @Autowired
  public void setGreetingProvider(@Qualifier(“hello") GreetingProvider greetingProvider){
     this.greetingProvider = greetingProvider;
  }
…
}
```

```
@Component // name is helloWorldGreetingProvider
@Qualifier(“hello”)
public class HelloWorldGreetingProvider
           implements GreetingProvider{
  …
}
```

greeting14

- `org.javaturk.spring.di.ch06.greeting.greeting14.Application`
  - `getBeans1()`
  - `GreetingProvider` has two implementations both of which are annotated by `@Component` and `@Qualifier`.
  - Observe how `@Qualifier` helps the injecton to resolve the bean.

Qualifier - IV

- Qualifiers can also be used in XML configuration file as a nested element `</qualifier>` of `</bean>`.
- `</qualifier>` has similarly one `String` attribute which is used when injecting with `@Autowired`.

greeting14

- `org.javaturk.spring.di.ch06.greeting.greeting14.Application`
  - `getBeans2()`
  - In XML file no classpath scanning is enabled so beans are resolved through the XML file but dependencies are resolved through `@Autowired`.
  - Observe how `</qualifier>` helps the injecton to resolve the bean.

Qualifying Beans

Custom Qualifier

Custom Qualifier

- Creating custom qualifiers might be a better solution than using `@Qualifier` with a `String` value.
- This can be done by defining a new annotation of type `@Qualifier`.

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

```
@Component
public class SelamGreetingProvider implements GreetingProvider{
   …
 @Autowired
  public void setGreetingProvider(@Selam GreetingProvider greetingProvider){
     this.greetingProvider = greetingProvider;
  }
}
```

greeting14

- `org.javaturk.spring.di.ch06.greeting.greeting14.Application`
  - `getBeans1()`
  - Remove all `@Qualifed`, `@Primary` annotations and `</qualifier>` elements in XML file.
  - Use `@Selam` and `@Hello` qualifiers for the beans that declare `@Component`.

Application

- `org.javaturk.spring.di.ch06.qualifier.app.Application`
    - This is another example for custom qualifiers.

Using Qualifiers

- Using any kinds of qualifier annotation invalidates qualification through naming.
- `@Qualifier` invalidates `@Primary` if used together
- A component may have more than one qualifier that are applied in different contexts.
- A component may both be a primary bean and have some other qualifiers.
  - It means the bean is selected in different contexts using different qualifiers.

Qualifying Beans

Qualification Through Generics

Qualification Through Generics

- **Spring** also allows qualification through generics.
- It is an implicit way of qualifying among candidate beans such as naming.

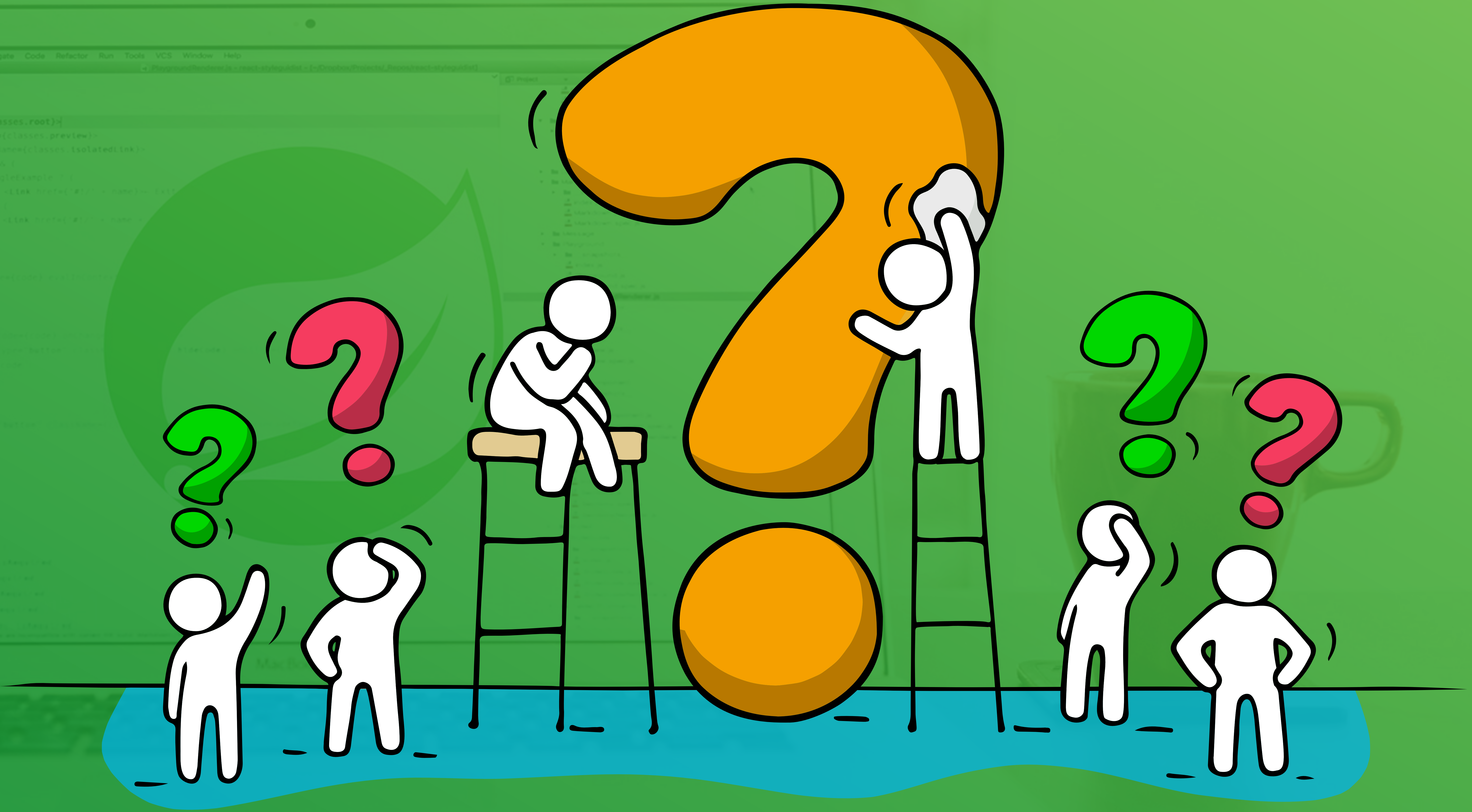
```
@Component
public class Person {

  @Autowired
  DeliveryPoint<HomeAddress> homeAddress;

  @Autowired
  DeliveryPoint<OfficeAddress> officeAddress;
  …
}
```

```
@Component
public class DeliveryPoint<Address> {

  private Address address;

  public Address getAddress() {
     return address;
  }

  public void setAddress(Address address) {
     this.address = address;
  }
}
```

Application

- `org.javaturk.spring.di.ch06.qualifier.generics.Application`

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

Qualifying Beans

@Profile

@Profile - I

- `org.springframework.context.annotation.Profile` is used to select components that are eligible for registration when one or more specified profiles are active.
- `Profile` is a logical group that has a name.
- `Profile` can be considered as a specific configuration of the beans for a specific purpose.
    - For different database servers,
    - For development, production or test environments etc.

@Profile - II

- `Profile` annotation has only one attribute called `value` which is of type `String` array.
- This is the same as using `profile` attribute of `</beans>` with profile names in XML file such as `<beans profile=“p1, p2">`.
- `Profile` annotation can be used as both at class level with `@Component` and `@Configuration` annotations and to create custom profiles.

@Profile - III

- `@Profile` has only one attribute, `value` which is of type `String` array.
  - So `@Profile` may have more than one profile name.
- `@Profile` may have simple name or more complicated expression that include to express some profile logic:
  - `!`: A logical “not” of the profile
  - `&`: A logical “and” of the profiles
  - `|`: A logical “or” of the profiles

@Profile - IV

- Default profile, `@Profile("default")` is the one that is enabled by default.
- If no profile is active, default profile is in effect.
- If any profile is enabled, the default profile does not apply.

Custom Profiles

- Custom profiles can be created using `@Profile` annotation.

Activating Profiles

- Profiles can be activated in several ways:
  - **`setActiveProfiles()`** method of **`org.springframework.core.env.ConfigurableEnvironment`**
  - setting **`spring.profiles.active`** property as JVM system property
  - using **`org.springframework.test.context.ActiveProfiles`** annotation
    - **`ActiveProfiles`** is mainly used in integration testing.

Application

- `org.javaturk.spring.di.ch06.profile.Application`

Qualifying Beans

@Conditional

@Conditional - I

- `org.springframework.context.annotation.Conditional` annotation is used to indicate that the bean is eligable to be registered only when the condition is valid.
- `@Conditional` became available in **Spring** 4.0.
- It is used for conditional registration of beans.
  - So it is like an if-else for the registration of beans.
- A **condition** is any state that can be determined programmatically before the bean definition is due to be registered.

@Conditional - II

- `@Conditional` can be used with `@Component` and `@Configuration`.
- `@Conditional` has an attribute of type array of `org.springframework.context.annotation.Condition` and its implementations so it can check one or conditions together.
    - All of the conditions should be valid in order for the bean to be registered.

Condition - I

- `Condition` interface represents a single condition that must be matched in order for a component to be registered.
- Conditions are checked immediately before the bean-definition is due to be registered.
- It has a method called `matches()` that receives a `ConditionContext`, and `AnnotatedTypeMetadata` parameters and returns a `boolean`.

Condition - II

- Any kind of condition can be considered for beans:
  - It can be whether a system or environmental property is available
  - It can be whether the bean has a specific annotation
  - etc.

Application

- `org.javaturk.spring.di.ch06.condition.Application`

@Condition vs. @Profile

- `@Condition` is a more generic version of `@Profile`.
- They both work on if-else fashion.
  - If the profile is correct or the condition is set then beans are registered.
- Profiles are mainly used to select among environmental configurations while conditions are used for any kind of selection of the beans.
- So profiles can be considered as a more coarse grained selection strategy than conditions.

selsoft

build better, deliver faster

Exercise

Exercise

- `org.javaturk.spring.di.ch06.ex.calculator.qualifier.Test`
- Use `@Component` and `@Autowired` for the beans and injections.
  - Create an XML configuration file with `</context:component-scan>` in it.
- Run `Test` and observe the application throws `NoUniqueBeanDefinitionException` inside `UnsatisfiedDependencyException`.

Exercise

- Solve qualification problem using three different ways:
  - Using naming convention
  - Using `@Qualifier`
    - Create an XML configuration file with `</context:component-scan>` in it.
  - Using `</qualifier>` in XML file which has all bean definitions and `<context:annotation-config />` declaration.

Scope

Singleton - Prototype

- Spring's components are singleton in default.
- This can be changed by using `@Scope` annotation.
- `@Scope` takes a `String` argument as the name of the scope.
- For singleton and prototype scope either `singleton` or `prototype` `String` values or the constants on `org.springframework.beans.factory.config.ConfigurableBeanFactory` can be used.
- Default value for `@Scope` is `singleton`.

ScopeExample

- `org.javaturk.spring.di.ch06.scope.ScopeExample`
    - Use the components defined in `greeting14` to change their scope.

Value Injection Using @Value

selsoft

build better, deliver faster

@Value - I

- `org.springframework.beans.factory.annotation.Value` is an annotation for external properties.
- It is used at field and parameter level.
- It is mostly used for expression-driven or property-driven dependency injection.
  - Expression-driven means using **Spring Expression Language** (**SpEL**),
  - Property-driven means accessing the properties of other beans.

SpEL - I

- **Spring Expression Language** (**SpEL**) is an expression language that provides querying and manipulating an object graph at runtime:
  - Literal expressions
  - Boolean and relational operators, assignment
  - Regular expressions
  - Class expressions and method invocation
  - Accessing properties, arrays, lists, and maps, etc.

SpEL - II

- Its API is mainly in `org.springframework.expression` and its sub packages.
- It has its own parser: `org.springframework.expression.ExpressionParser` which has an implementation `SpelExpressionParser`.
- Every SpEL expression is represented by `Expression` interface.

SpelExample

- `org.javaturk.spring.di.ch06.spel.SpelExample`

@Value - II

- `@Value` has a required attribute called `value` which designates the value.
- Simple string values, properties of other beans and more complex values using SpEL can be injected.
  - All type conversions are handled automatically by **Spring**.

@Value - III

- A SpEL element defines the value of the property of a bean using **`#{expression}`**.
  - For **`@Value`** annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.
  - getter methods are called when accessing bean properties.
- A SpEL element defines the value of a property specified in a properties file through **`${property-name}`**.

Properties File - I

- Properties files are specified using `<context:property-placeholder location="">` in XML file or `@PropertySource` in source code.
- If both are specified **Spring** combines them.
- If a property name collision occurs the last source overrides.
  - If both are specified properties file specified in XML file is loaded first and then properties file specified in annotation is loaded.

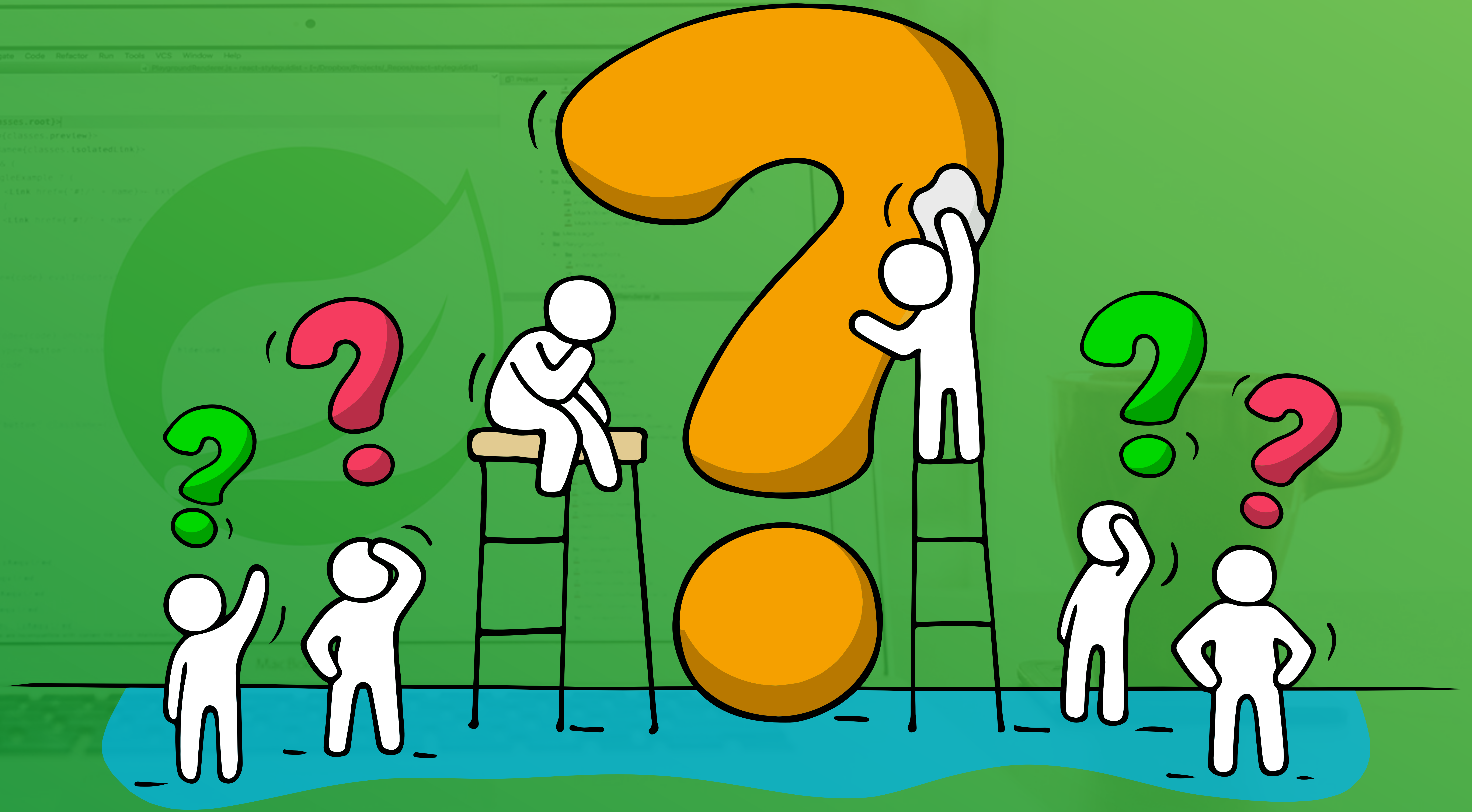
Properties File - II

- Both use `classpath:` and `file:` for the path of the properties file.
- `${…}` placeholders can also be used in location of properties file when specified in XML or annotation for the replacement of some values.
- In case of multiple properties file `@PropertiesResources` annotation can be used.
- In XML multiple properties files can be provided using comma-separated paths.

ValueExample

- `org.javaturk.spring.di.ch06.value.ValueExample`

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr

XML vs. Annotations for Configuration

selsoft

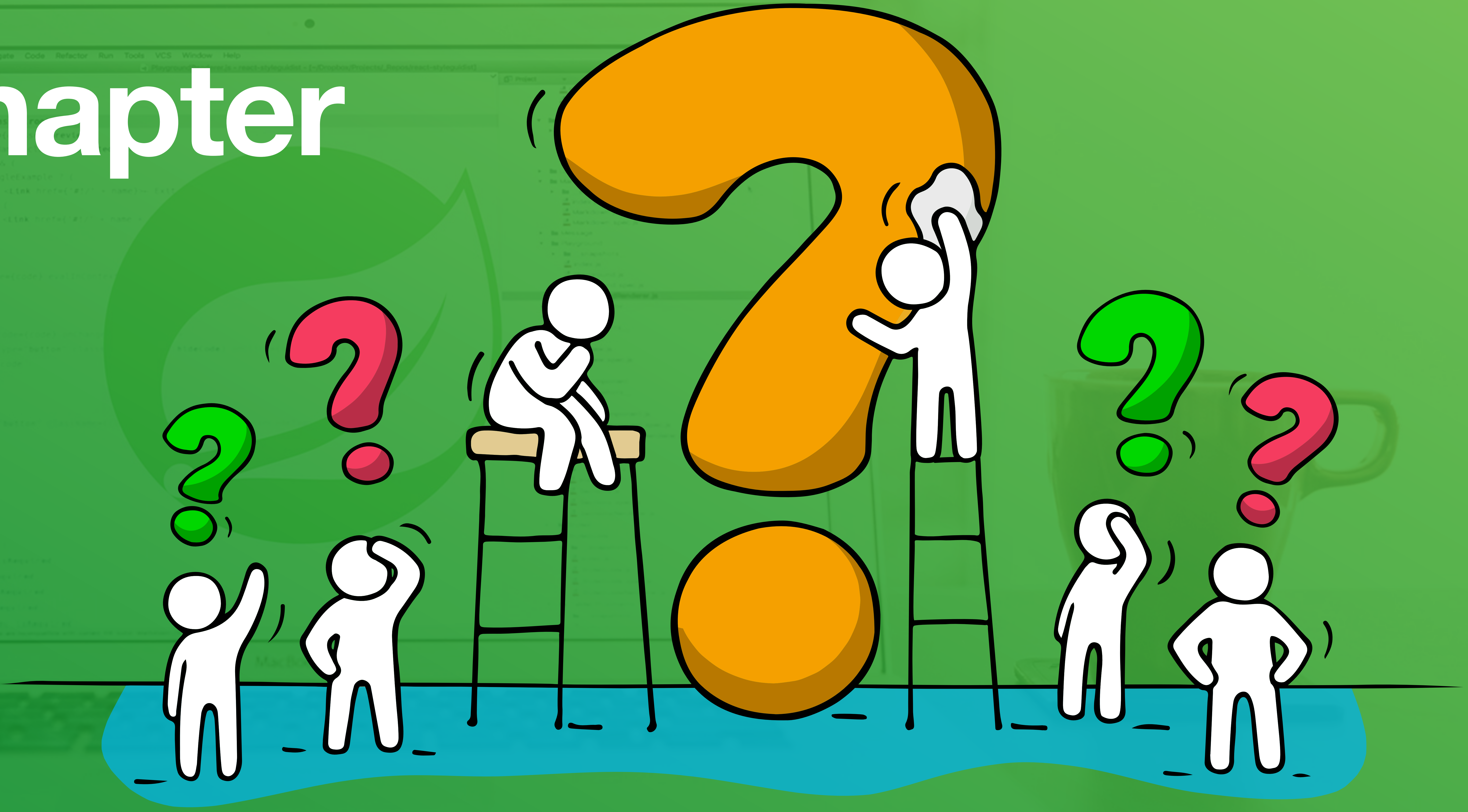
build better, deliver faster

XML vs. Annotation Metadata

- Using XML files or annotations for configuration metadata has its own advantages and disadvantages.
- XML files can get bigger easily which causes complexity but saves source code from configuration info leaving all beans as POJOs.
- Annotations provide small and contextual information regarding beans making them developer-friendly but they make source code depended on annotations which are part of **Spring** and modification necessary when configuration changes.

End of Chapter

Time for Questions!



selsoft
build better, deliver faster

info@selsoft.com.tr
selsoft.com.tr