selsoft
build better, deliver faster

Enterprise Application Development with Spring

Chapter 5: XML Configuration

Instructor
Akın Kaldıroğlu
Expert for Agile Software Development and Java

Topics

- **Inversion of Control (IoC)**
  - History of IoC
- **Spring IoC Container**
  - Main Objects
  - Configuration Metadata
  - Specifying Dependencies
  - Autowiring Dependencies

XML Configuration

selsoft

build better, deliver faster

selsoft

build better, deliver faster

XML Configuration

XML Configuration Metadata - I

- Using XML files for configuration metadata is the classical way of defining the application context.
- The structure of XML files for configuration is determined by a schema that is reachable at https://www.springframework.org/schema/beans/spring-beans.xsd.
- The schema provides information on its structure and elements.
- So an XML file for **Spring** configuration must conform to this schema.
  - Name of the configuration file does not matter, its path is used.

XML Configuration Metadata - II

- XML-based configuration metadata configures beans as `<bean/>` elements inside a top-level `<beans/>` element.
  - Beans refer to each other for dependencies.
- More than one XML configuration file for an application can be used.
- XML configuration files are passed to the constructors of the `ApplicationContext` implementations such as `ClassPathXMLApplicationContext` and `FileSystemXMLApplicationContext` to be loaded into the application.

XML Configuration Metadata - III

- Root element in an XML configuration file must be `<beans/>`.
- An XML configuration file can have only one top-level `<beans/>` declaration, which's the rule of XML.
- An XML configuration file can have nested `<beans/>` declarations, i.e. `<beans/>` inside `<beans/>` in order to have subset bean definitions.
  - In the case of nested `<beans/>`, `<beans/>` declaration must be the last one in outer `<beans/>` declaration otherwise parsing error happens.
  - So nothing after nested `<beans/>` declaration is allowed.

XML Configuration Metadata - IV

- More than one XML configuration file can be used to properly organize bean configurations.
  - DAOs, backing beans, transactions can or should all be defined in different files.
- For this purpose one or more `<import/>` is used inside `<beans/>`.

```xml
<beans>
   <import resource="services.xml"/>
   <import resource="resources/messageSource.xml"/>
   <import resource="/resources/themeSource.xml"/>

   <bean id="bean1" class="..."/>
   <bean id="bean2" class="..."/>
</beans>
```

selsoft

build better, deliver faster

</bean>

<bean/>

- `<bean/>` is typically declared with three attributes:
  - `id` and `name` are a string to identify a bean definition.
  - The `class` defines the type of the bean using fully qualified name of its class.

```
<?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
  https://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="..." name="…" class="…">
     <!-- collaborators and configuration for this bean -->
   </bean>

   <bean id="..." name="…" class="...">
     <!-- collaborators and configuration for this bean -->
   </bean>
  …
</beans>
```

id Attribute

- `id` must be unique within a `<beans/>` and must have only one value.

```
<beans>
  <bean id="beanA"
    class="org.javaturk.spring.di.ch03.domain.BeanA"/>
</beans>
```

```
public class BeanA {
  …

}
```

- Before **Spring** 3.0 it is used to be ID of XML but now it is a string (so it can start with a digit and can have special characters now) and enforced by **Spring** IoC not by XML parser.

```
<beans>
  <bean id="5-beanA*"
    class="org.javaturk.spring.di.ch03.domain.BeanA"/>
</beans>
```

name Attribute

- `name` is a string and used to identify a bean and may have more than one value delimited by a comma “,”, a semicolon “;” or a space “ ”.
    - So a bean may have more than one `name`.
- All names are aliases to the bean and can be used to refer to the same bean in different contexts of large systems.
- `id` can be considered as a default name for the bean.

```
<beans>
  <bean id=“beanA” name=“BEAN_A, bean_a; bean_A BEAN-A”
    class="org.javaturk.spring.di.ch03.domain.BeanA">
  </bean>
</beans>
```

alias Element - I

- **`alias`** is a string element and used to provide aliases to a bean.
- **`alias`** is not an attribute, it is an element and is declared using one of the **`name`**s of the bean.
- Each **`alias`** provides a new **`name`** to the bean and takes only one value.

```
<beans>
  <bean id="beanA" name="BEAN_A, bean_a; bean_A BEAN-A"
    class="org.javaturk.spring.di.ch03.domain.BeanA"/>

  <alias name="beanA" alias="a"  />
  <alias name="beanA" alias="aa" />
</beans>
```

alias Element - II

- `alias` can be defined in a different XML resource.
- In fact if different parties of a project want to refer to the same bean declaration with different names they would create aliases most probably in different XML configuration files of the same application.

```
<beans>
  <bean id="beanA" name="BEAN_A, bean_a; bean_A BEAN-A"
    class="org.javaturk.spring.di.ch03.domain.BeanA"/>
</beans>
```

```
<beans>
  <alias name="beanA" alias="a"  />
  <alias name="beanA" alias="aa" />
</beans>
```

Bean Identifiers - I

- In XML configuration **`id`**, **`name`** and **`alias`** are used as identifiers.
- **Spring** first looks for an **`id`** if not found looks for the first of **`name`**s.
    - In fact **`id`** and **`name`** are not required, in which case the beans can't be queried by them, they can be reachable only by type.
    - If they are not supplied **Spring** produces a name for the bean using its wholly qualilied class name.

```
<beans>
  <bean class="org.javaturk.spring.di.ch03.domain.BeanA"/>
</beans>
```

```
BeanA beanA = (BeanA) context.getBean("org.javaturk.spring.di.ch03.domain.BeanA");
```

Bean Identifiers - II

- But if multiple beans of the same type are defined without an `id` or `name`, `org.springframework.beans.factory.NoSuchBeanDefinitionException` is thrown when either initializing `ApplicationContext` or injection.

- In case of classpath scanning, if names are not supplied for the components **Spring** produces names for them using the class name whose first letter is converted to lower case.

Bean Identifiers - III

- All **`id`**s, **`name`**s and **`alias`**es in the same **`<beans/>`** must be unique otherwise **`org.springframework.beans.factory.parsing.BeanDefinitionParsingException`** is thrown.
- **`id`**, **`name`** and **`alias`** can only be reused in different **`<beans/>`** in which case later definition overrides previous definition.
  - This may cause ambiguities.
- Beans in different **`<beans/>`** elements can refer to each other.

Bean Identifiers - IV

- For example, when a nested `<beans/>` element reuses the same `id`, `name` and `alias` value, **Spring** may throw a `ClassCastException` if the type of the bean is changed.

```
<beans>
  <bean id="beanA"
    class="org.javaturk.spring.di.ch03.domain.BeanA"/>
  <beans>
   <bean id="beanA"
     class="org.javaturk.spring.di.ch03.domain.BeanB"/>
  </beans>
</beans>
```

```
// Throws ClassCastException
BeanA beanA = (BeanA) context.getBean("beanA");
// That's ok
Object objectBeanA = context.getBean("beanA");
```

Bean Identifiers - V

- Moreover when **`id`** is reused to identify a new bean then all **`name`**s and **`alias`**es belonging to that **`id`** also change to refer to the new bean.
- This is true only when **`id`** is reused, if only a **`name`** or a **`alias`** is reused it does not effect anything becaue **`id`** is the main vehicle to identify a bean.
- And reusing causes deep problems!

```
<beans>
  <bean id="beanA" name=“xxx yyy”
        class=“org.javaturk.spring.di.ch03.domain.BeanA"/>
  </beans>
   <bean id="beanA"
      class=“org.javaturk.spring.di.ch03.domain.BeanB"/>
  </beans>
</beans>
```

```
// What are the types of the “object”?
Object object = context.getBean("xxx");
System.out.println(object);
object = context.getBean(“yyy");
System.out.println(object);
```

IdAndNameExample

- `org.javaturk.spring.di.ch05.bean.id.IdAndNameExample`

IdAndNameExample2

- **`org.javaturk.spring.di.ch05.bean.id.IdAndNameExample2`**
  - In this example notice how reusing the same **`name`** and **`alias`** in nested **`<beans/>`** causes problems.
  - Notice how beans in different **`<beans/>`** refers to each other.

Bean Identifiers - VI

- So every bean has one or more identifiers.
- Use meaningful and lower camel case identifiers for beans.
- And provide only one identifier as long as it suffices to identify the bean in different contexts and avoiding the reuse of **`id`**, **`name`** or **`alias`** is best practice.
- In case of more than one identifier for a bean the extra ones are aliases.
  - All **`name`**s and **`alias`**es are pointers to refer to the same bean in different contexts of large systems.

class Attribute

- Type of the bean that is specified in the `class` attribute must be a class unless it is specified as `abstract` in `<bean/>`.
    - Otherwise **Spring** throws `org.springframework.beans.BeanInstantiationException` exception.
- If `true` is provided for the `abstract` attribute of `<bean/>` then **Spring** does not create an instance of this bean.

Other Attributes of <bean/>

- Other attributes of `<bean/>` element are
  - `parent`
  - `abstract`
  - `scope`
  - `lazy-init`
  - `autowire`

</bean>

Bean Inheritance

Inheritance in Bean Definition - I

- Inheritance relationship among beans can be defined using `parent` attribute of `<bean/>`.
- `parent` attribute takes one identifier of the parent class as value.
- The purpose of this is to allow the child bean to inherit bean definition from its parent.
- If the child bean does not provide any property it inherits what is defined for its parent if `parent` attribute exists.

Inheritance in Bean Definition - II

- The child class can override the bean definition of its parent by providing its own properties as well.

InheritanceExample1

- `org.javaturk.spring.di.ch05.bean.inheritance.InheritanceExample1`

Abstract Bean Definition

- A bean can be declared as `abstract` using `abstract` attribute of `<bean/>`.
- In this case **Spring** does not create the instance of the `abstract` bean.
- Trying to get the instance of such bean throws `org.springframework.beans.factory.BeanIsAbstractException`.
- A bean can be declared as `abstract` in XML configuration even though it is not an `abstract` class and **Spring** never creates its instance.

InheritanceExample2

- `org.javaturk.spring.di.ch05.bean.inheritance.InheritanceExample2`

</bean>

Loading Beans

Loading Issues

- There are mainly two issues regarding the loading of beans:
  - Eager loading vs. lazy loading is about when the instances of the beans are created.
  - Singleton vs. prototype is about how many instances of a bean are created.
- IoC container has default behaviors on these issues but they can be modified.

</bean>

Loading Beans

Singleton vs. Protoype Beans

Singleton-Prototype Beans - I

- When **`ApplicationContext`** creates a bean it configures it as a singleton bean which has only one instance.
- This is an issue of scope which can be specified by the **`scope`** attribute of the **`<bean/>`** element.
- Its implicit default value is **`singleton`** and can be controlled by giving one of different acceptable **`String`** values such as **`prototype`**.
- In **`prototype`** bean declaration a new bean instance is created each time it is requested from **`ApplicationContext`**.

Singleton-Prototype Beans - II

- When a bean is defined as singleton `ApplicationContext` creates its instance eagerly unless it is specified for lazy initialization.
    - We'll see eager and lazy initialization soon.
- But when a bean is defined as non-singleton such as `prototype` `ApplicationContext` creates its instance only when the application tries to fetch it.

greeting08

- **`org.javaturk.spring.di.ch05.greeting.greeting08`**
  - Give the value **`prototype`** to the **`scope`** attribute on the **`<bean/>`** element and fetch the same bean from the **`ApplicationContext`** repeating times in **`checkScope()`** method.

Singleton vs. Prototype Beans - I

- If a bean is defined as a singleton then it can have at most one instance and that instance is shared with all other beans into which it is injected.
    - All beans that receive the singleton collaborator bean via injection have the same instance, which becomes the shared state among the beans.
- If a bean is defined as a prototype then every bean into which it is injected has its own instance of the collaborator bean.
    - In this case the instances of the collaborator bean is not shared, every instance of the collaborator bean is part of the private state of a bean it is injected.

Singleton vs. Prototype Beans - II

- So when to use singleton and when to use prototype?
- Of course in terms of run-time efficiency less objects would be much better so the singleton would be the choice.
- But most of the time the run-time efficiency is not the unique factor that determines the choice.
- The key term on this issue is the state, i.e. whether the collaborator bean has a state and if it does then whether that state can be changed by the beans into which it is injected and the environment is multi-threaded.

Singleton vs. Prototype Beans - III

- If the collaborator bean does not have a state, i.e. it is stateless not statefull, it shoud be a singleton.
    - All method calls made to the collaborator bean can only use the local variables which are thread-safe.
- If the collaborator bean does have a state, i.e. it is statefull not stateless then question would be whether the beans want to change that state:
    - If the state of the collaborator bean is immutable i.e. only readable not writable then it shoud be a singleton too.

Singleton vs. Prototype Beans - IV

- If the collaborator bean has a mutable state then question becomes whether the environment is multi-threaded or not.
- Beans created by **Spring** are not thread safe.
  - If the collaborator bean has mutable state then sharing it in a multi-threaded environment requires synchronization to avoid race conditions and corruption in the state of the singleton.
  - In this case using prototype beans could be more appropriate due to the fact that creating prototype beans can be more efficient in terms of both run-time efficiency and code complexity .

Singleton vs. Prototype Beans - V

- In the case of prototype all other beans would have its own copy of the collaborator bean and will be able to change its state freely without any need for synchronization.
- That means the state of the collaborator bean would be part of the private state of all beans into which the collaborator bean is injected.

- On the other hand in a multi-threaded environment synchronization on a singleton instance leads to a more complex code while using prototype instances without any need for synchronization produces simpler code.

Singleton vs. Prototype Beans - VI

- Think about a calculator bean injected into many different clients that would want the calculator bean to make calculations.
  - Totally stateless calculator or unmutable state => singleton
  - Statefull calculator in a single-threaded environment => singleton
  - Statefull calculator in a multi-threaded environment:
    - Singleton => Synchronization of the methods that change the state
    - Prototype => No need for synchronization. Prototype bean is part of each client's private state

</bean>

Loading Beans

Eager vs. Lazy Loading

Eager and Lazy Loading - I

- `ApplicationContext` by default creates beans eagerly during the initialization of **Spring** IoC container.
- To change this behavior the `lazy-init` attribute of the `<bean/>` is used.
  - Its implicit default value is `true` and can be set to `true` or `false` values.
- Lazy-initialization at the container level can be configured by using the `default-lazy-init` attribute on the `<beans/>` element.
- Specifying the `lazy-init` attribute on the `<bean/>` element for a specific bean overrides the behavior of the `<beans/>` element.

Eager and Lazy Loading - II

- On the other hand when a bean is created whether eagerly or lazily, all its collaborators are also created.
- When a lazy-initialized bean is a dependency of a bean that is not lazy-initialized, the `ApplicationContext` creates the lazy-initialized bean at startup to satisfy the dependencies.
- So eagerly-initialized beans causes their lazily-initialized dependents to be loaded eagerly.

Eager and Lazy Loading - III

- Declaring all beans to be loaded eagerly may cause the bootstrap of the application to take some time but all beans instances would be ready to use when the application finishes starting.
  - This is true for singleton beans.
  - For prototype beans all creations are made lazily so if the application does not ask for a bean its instance is not created.
  - This is true even though prototype beans are declared to be loaded eagerly, they are always created when asked from the context.

Eager and Lazy Loading - IV

- So specifying lazy initialization for singleton beans can be considered a performance tuning issue for the start up of the applications.

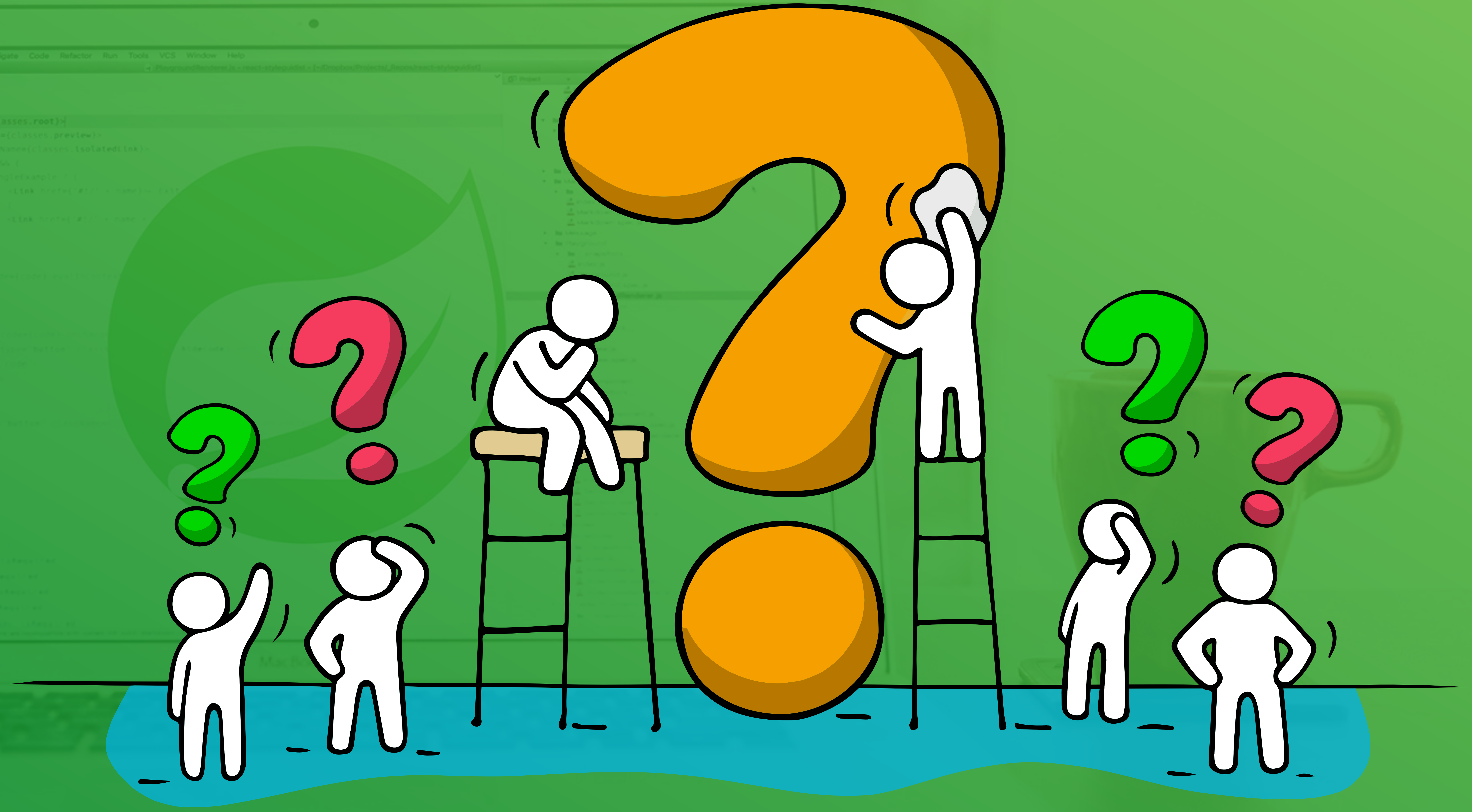
LazyExample

- `org.javaturk.spring.di.ch05.lazy.LazyExample`
  - Do not fetch any bean from context object just to see the default behavior for both singletons and prototypes.
  - Then use **`default-lazy-init`** attribute on the **`<beans/>`** element.
  - **`lazy-init`** attribute on the **`<bean/>`** element can be used to override the behavior of the **`<beans/>`** element.

greeting08

- **`org.javaturk.spring.di.ch05.greeting.greeting08`**
  - Do not fetch any bean from **`ApplicationContext`** object just to see the default eager behavior.
  - Then use **`default-lazy-init`** attribute on the **`<beans/>`** element.
  - Then use **`lazy-init`** attribute on the **`<bean/>`** element to override the behavior of the **`<beans/>`** element.

Time for Questions!



selsoft
build better, deliver faster

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

</bean>

Bean Instantiation

Bean Instantiation - I

- **Spring** creates the instances of the declared beans using reflection.
    - Depending on the injection of the dependencies, default, non-argument constructor or a suitable argument-constructor must be provided.
- **Spring** either invokes the constructor of the bean or it calls a static factory method on the same class if specified to create the instance.

```
<beans>
  <bean id="beanA" name="bean_A, bean_a, BEAN_A"
         class="org.javaturk.spring.di.ch03.domain.BeanA">
  </bean>

  <alias name="beanA" alias="a"/>
  <alias name="beanA" alias="aa"/>
</beans>
```

```
public class BeanA {
 …

}
```

Bean Instantiation - II

- **Spring** can call a static factory method to create the instance.
- For the purpose the name of the static factory method must be provided for the `factory-method` attribute of `<bean/>` element.
- **Spring** still injects all declared dependencies to the bean.

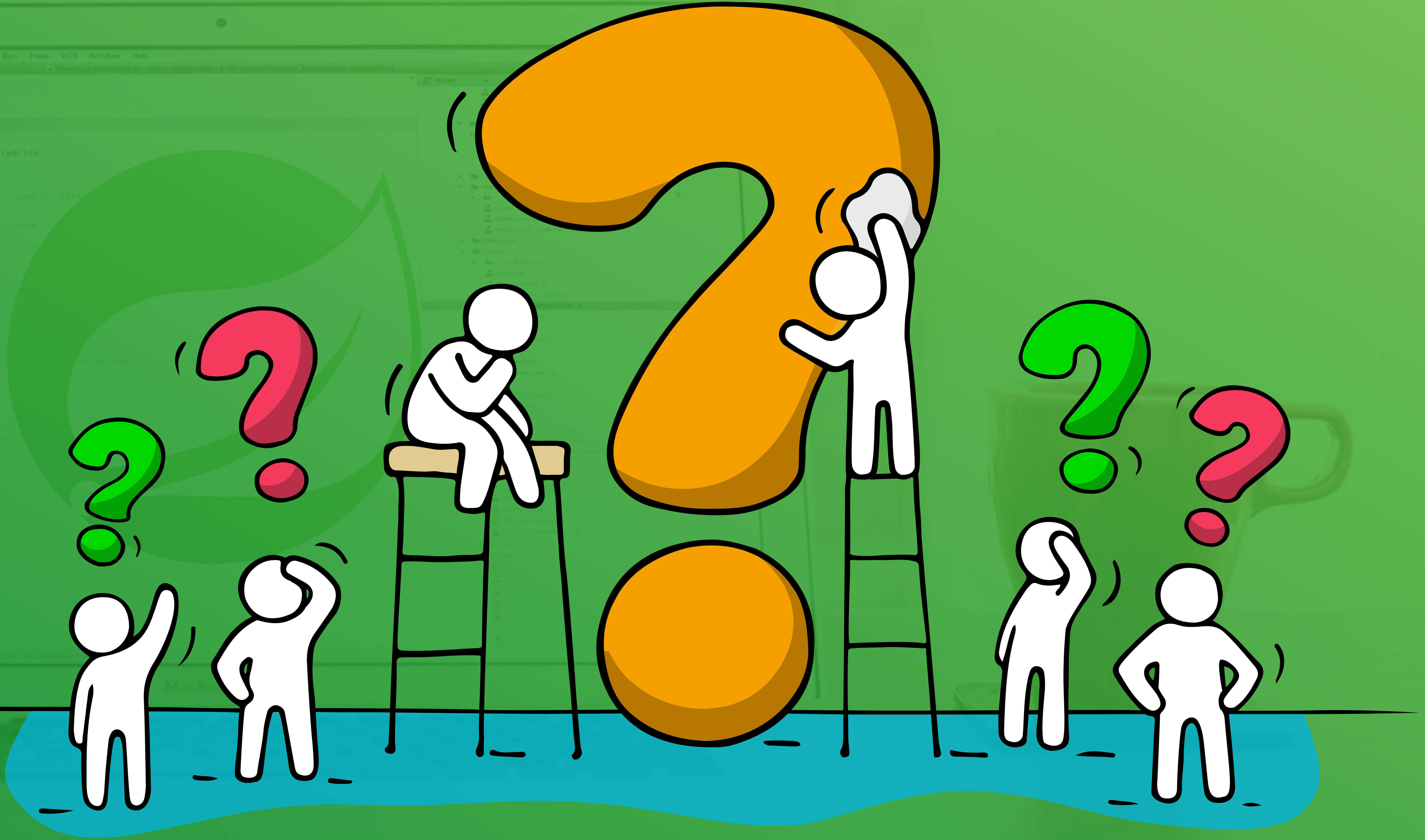
Bean Instantiation - III

- **Spring** can call an instance factory method on another bean to create the instance.
- For the purpose the name of the class that has the non-static factory method along with the bean name must be provided for the **`factory-bean`** and **`factory-method`** attributes of **`<bean/>`** element.
  - **`class`** attribute can be omitted.
- **Spring** still injects all declared dependencies to the object.

FactoryExample

- `org.javaturk.spring.di.ch05.bean.factory.FactoryExample`
  - `useStaticFactoryMethod()`
  - `useInstanceFactoryMethod()`

Time for Questions!



selsoft
build better, deliver faster

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

</bean>

Specifying Dependencies

Dependency Definition

- There are mainly two ways to define dependencies in the XML configuration file:
    - Explicit definition through constructor and property injection
    - Implicit definition through **`autowire`** attribute of **`<bean/>`** element

</bean>

Specifying Dependencies

Constructor & Property Injection

Constructor and Property Injection - I

- In dependency injection all collaborators are injected into the bean on which it depends on.
- In XML-based configuration there are mainly two types of explicit injection:
  - Constructor injection
  - Property or setter injection
- Both ways use either `ref` attribute to refer to another bean or `value` attribute for a primitive value or a `String`.

Constructor and Property Injection - II

- For constructor injection `<constructor-arg/>` and for setter (or property) injection `<property/>` is used.
- In case of injecting a bean using `ref` attribute either `id` or one of the `name`s or `alias`es of the collaborator bean can be used inside `<constructor-arg/>` or `<property/>` element.
  - A nested `<ref bean='...'/>` can also be used.
- To inject a literal value such as a primitive or a String `value` attribute is used inside `<constructor-arg/>` or `<property/>` element.

Constructor and Property Injection - III

- To inject a `null` value for beans `<null/>` element is used in `<constructor-arg/>` and `<property/>` elements.

Constructor Injection - I

- For constructor injection `<constructor-arg/>` is used.
- Collaborator bean is referred by the `ref` element inside `<constructor-arg/>` which receives the `id`, `name` or `alias` of the bean.
  - Or nested `<ref bean='...'/>` is used.
- The bean must have a constructor that receives an instance of the collaborator bean as an argument otherwise **Spring** throws `org.springframework.beans.factory.UnsatisfiedDependencyException`.

Constructor Injection - II

- Resolution of the constructor argument occurs basically by using the argument's type.
- In case of more than one argument **`index`**, **`name`** and **`type`** attributes can be used to help to resolve dependencies:
  - **`index`** shows the index of the argument and takes an integer starting from 0
  - **`name`** takes a string as the name of the parameter the constructor receives
  - **`type`** takes a string as the type of the parameter the constructor receives

Constructor Injection - III

- Using `index` would be the simplest way to help **Spring** IoC in resolving the arguments.

Property Injection - I

- For property injection **`<property/>`** is used.
- **`name`** attribute of the **`<property/>`** shows the name of the property.
- Collaborator bean is referred by the **`ref`** element inside **`<property/>`** which receives the **`id`**, **`name`** or **`alias`** of the bean.
  - Or nested **`<ref bean='...'/>`** is used.
- Resolution of the argument occurs basically by JavaBeans naming convention in the setter method.

Property Injection - II

- The bean must have a properly-named setter method that receives an instance of the collaborator bean as an argument otherwise **Spring** throws **`org.springframework.beans.factory.UnsatisfiedDependencyException`**.

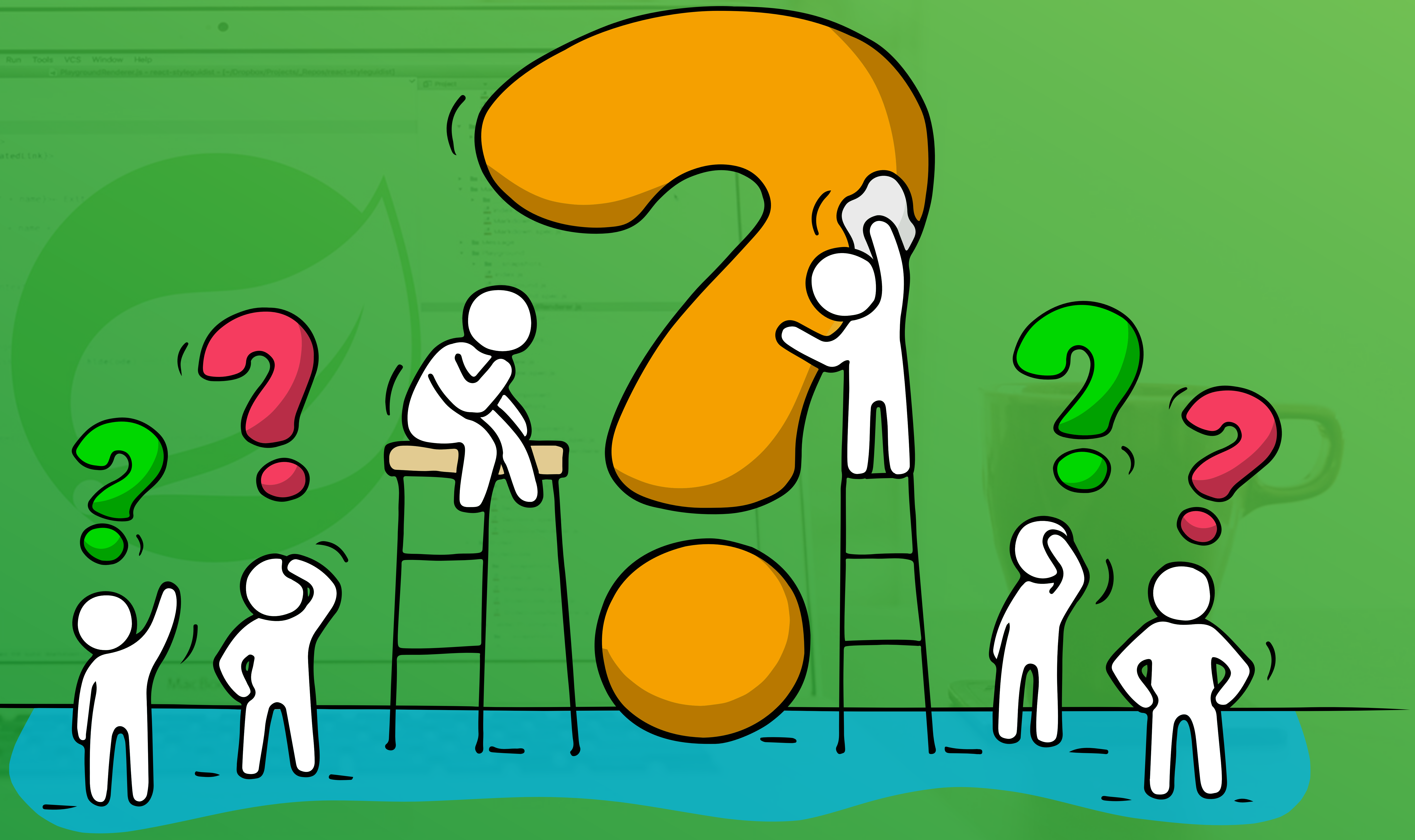
Inner Bean

- Sometimes a dependency is defined directly using `<bean/>` tag without an `id` or `name` in which case the bean is called **inner bean**.
- In this case the `class` attribute of the `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements takes as value the class of the bean.

greeting09

- **`org.javaturk.spring.di.ch05.greeting.greeting09`**
  - Notice constructor and setter injections using **`id`**, **`name`** or **`alias`** to refer to beans.

Time for Questions!



selsoft

build better, deliver faster

info@selsoft.com.tr

selsoft.com.tr

</bean>

Specifying Dependencies

Value Injection

Value Injection - I

- Values like beans can be injected into constructors and setter methods.
- For this purpose instead of **`ref`** attribute or **`<ref bean=“…”>`** element to refer to a bean, **`value`** attribute is used in **`<constructor-arg/>`** or **`</property>`** element.
    - A nested **`<value>`**…**`<value/>`** can also be used.
- **`value`** attribute is used to pass primitives, **`String`** literals and collections (array, list, set and map) of those types.

Value Injection - II

- All type conversions are made automatically by **Spring**.
- Injecting collections will be handled soon.

Value Injection - II

- Of course proper constructor and setter methods must be provided otherwise **Spring** throws `org.springframework.beans.factory.UnsatisfiedDependencyException`.

```
<bean id="constructorWithIndex1" class="….ValueInjection1">
     <constructor-arg index="0" value="0" />
     <constructor-arg index="1" value="1" />
     <constructor-arg index="2" value="3.14" />
     <constructor-arg index="3" value="true" />
     <constructor-arg index="4" value="Selam" />
     <constructor-arg index="5" value="Hello" />
</bean>
```

```
<bean id="property1" class="….ValueInjection1">
     <property name="i" value="0" />
     <property name="j" value="1" />
     <property name="d" value="3.14" />
     <property name="b" value="true" />
     <property name="s1" value="Selam" />
     <property name="s2" value="Hello" />
</bean>
```

```
public class ValueInjection1 {
 public ValueInjection1(int i, int j, double d, boolean b, String s1, String s2) {…}

 public void setI(int i) { this.i = i;}
 public void setJ(int j) { this.j = j;}
 public void setD(double d) { this.d = d;}
 public void setB(boolean b) { this.b = b;}
 public void setS1(String s1) { this.s1 = s1;}
 public void setS2(String s2) { this.s2 = s2;}
 …
}
```

</bean>

Specifying Dependencies

Collection Injection

Collection Injection - I

- Collections of beans or values can also be injected into constructors and setter methods.
- As a collection array, **`List`**, **`Set`** and **`Map`** implementations can be used.
- To specify collections in XML file **`</array>`**, **`</list>`**, **`</set>`**, and **`</map>`** are used.
  - In case of values all type conversions are made automatically by **Spring**.

Collection Injection - II

- While array, `List`, `Set` takes values or references to beans `Map` receives key-value pairs which is called entry.
- So `</array>`, `</list>`, `</set>` elements use nested `<value>`… `<value/>` for values or nested `<ref bean=‘...'/>` for beans.
- `</map>` uses `</entry>` element to pass key-value pairs.

Collection Injection - III

- `</entry>` element in `</map>` has several attributes:
  - **`key`**: Used for key of the entry if it is value
  - **`key-ref`**: Used for key or value of entry if it is another bean
  - **`value`**: Used for value of entry if it is value
  - **`value-ref`**: Used for key or value of entry if it is another bean
- Nested `<ref bean='...'/>` can also be used for beans.

```xml
<bean id="constructor1" class="o….ValueInjection2">
     <constructor-arg>
          <array>
               <ref bean="beanA" />
               <ref bean="beanA2" />
               <ref bean="5-beanA*" />
          </array>
     </constructor-arg>
     <constructor-arg>
          <list>
               <ref bean="beanB" />
          </list>
     </constructor-arg>
     <constructor-arg>
          <set>
               <ref bean="beanC" />
          </set>
     </constructor-arg>
     <constructor-arg>
          <map>
               <entry key="1" value-ref="beanA" />
               <entry key="10" value-ref="beanA2" />
               <entry key="100" value-ref="5-beanA*" />
          </map>
     </constructor-arg>
     <constructor-arg>
          <map>
            <entry key="1"><ref bean="beanA" /></entry>
            <entry key="10"><ref bean="beanA2" /></entry>
            <entry key="100"><ref bean="5-beanA*" /></entry>
          </map>
     </constructor-arg>
</bean>
```

```java
public class ValueInjection2 {
  private BeanA[] array;
  private List<BeanB> list;
  private Set<BeanC> set;
  private Map<Integer, BeanA> map1;
  private Map<Integer, BeanA> map2;

  public ValueInjection2(BeanA[] array, List<BeanB> list,
                         Set<BeanC> set, Map<Integer, BeanA> map1,
                         Map<Integer, BeanA> map2) {
 …
}
```

Collection Injection - III

- For bean references `<null/>` can be used
- In `</entry>` element of `</map>` key can be given `null` value using `<key> <null/> </key>` and value can be given `null` value using only `<null/>`
  - Don't forget `</entry>` element doesn't take two `<null/>` elements such as `<entry> <null/> <null/></entry>`. for both key and value.
  - Use instead: `<entry> <key> <null/> </key> <null/></entry>`

ValueInjectionExample

- `org.javaturk.spring.di.ch05.injection.value.ValueInjectionExample`

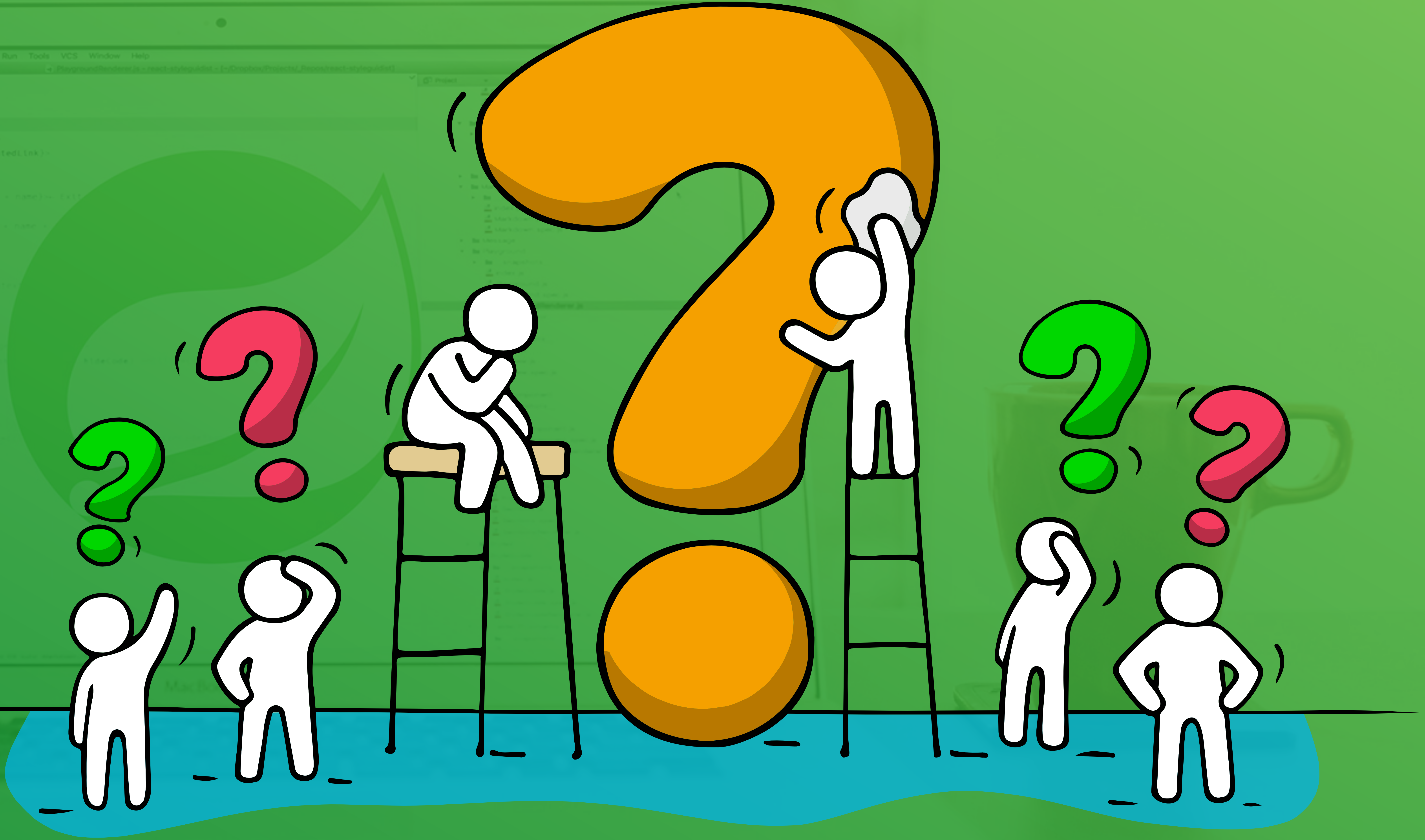
CollectionInjectionExample2

- `org.javaturk.spring.di.ch05.injection.collection.CollectionInjectionExample`

Constructor vs. Property Injection

- The constructor injection over the property injection can be preferred when the bean should be immutable.
- On the other hand constructor injection should be used for mandatory dependencies and to avoid `NullPointerException` exceptions.

Time for Questions!



selsoft
build better, deliver faster

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

selsoft

build better, deliver faster

Exercise

Exercise

- `org.javaturk.spring.di.ch03.ex.calculator.collection.Test`
- Create an XML configuration file for the `Test`.

selsoft

build better, deliver faster

</bean>

Specifying Dependencies

autowire

autowire - I

- `autowire` attribute of the `<bean/>` element allows **Spring** IoC container to find and satisfy dependencies automatically.
  - Depended beans must still be defined in the XML file.
- This way provides cleaner XML configuration files because `<constructor-arg>` and `<property>` tags are not needed.
- All dependencies are resolved through the conventions specified by `autowire` attribute value.

autowire - II

- `autowire` attribute can have one of three values:
  - **constructor**: Allows to inject into constructor.
  - **byName**: Allows to inject into a property resolving by name of a setter method.
  - **byType**: Allows to inject into a property resolving by type of the argument that is received by a method.
- **byName** and **byType** are property injections but the first one looks for a proper setter method while the second one looks for any method with a proper type.

autowire - II

- `autowire` attribute can have one of three values:
  - `constructor`: Allows to inject into constructor.
  - `byName`: Allows to inject into a property resolving by name of the setter method.
  - `byType`: Allows to inject into a property resolving by type of the argument that is received by a method.
- Its default value is `no`.

autowire - III

- Using `autowire` attribute has some limitations:
  - Primitives and `String` objects can not be injected,
  - Autowiring may create ambuguities.
- So explicitly specifying dependencies in the XML file would avoid these problems.

autowire-candidate - I

- Beans can be excluded from autowiring by setting the `autowire-candidate` attribute of the `<bean/>` element to `false` in XML configuration file.
- The container doesn’t makes that specific bean definition available to the autowiring.

autowire-candidate - II

- But note that **`autowire-candidate`** attribute only affects type-based autowiring.
- It doesn’t affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate.
    - So autowiring by name nevertheless injects a bean if the name matches.

greeting10

- `org.javaturk.spring.di.ch05.greeting.greeting10`

Some Other Issues

selsoft
build better, deliver faster

Others - I

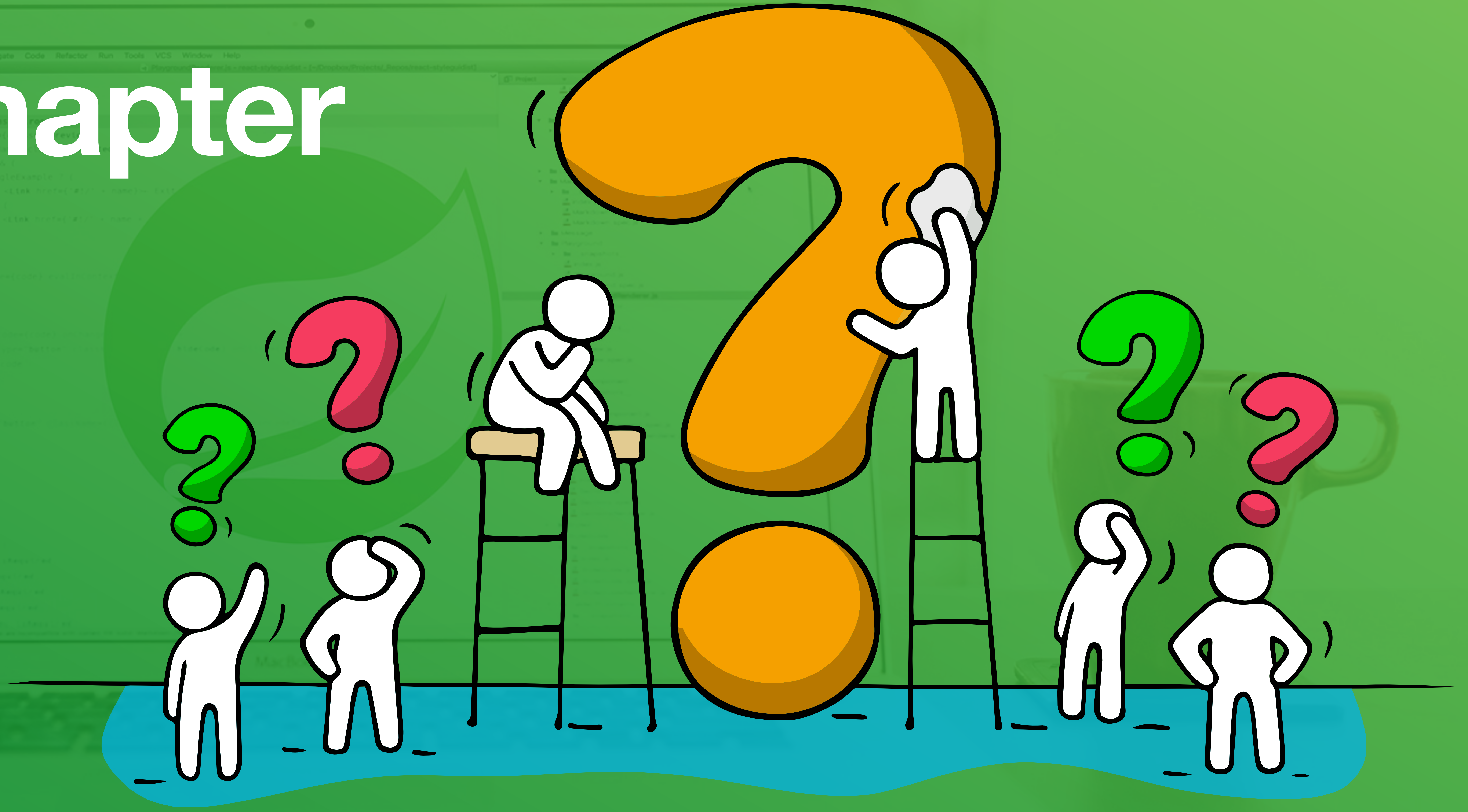
- There are some other issues for beans and dependencies:
    - Use of `p-namespace` and `c-namespace`

Others - II

- How to let a singleton bean to have a prototype instances of another bean every time it needs to?
    - This can be handled by implementing **`ApplicationContextAware`** interface.

End of Chapter

Time for Questions!



selsoft
build better, deliver faster

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