selsoft
build better, deliver faster

Enterprise Application Development with Spring

Chapter 8: Bean Lifecycle

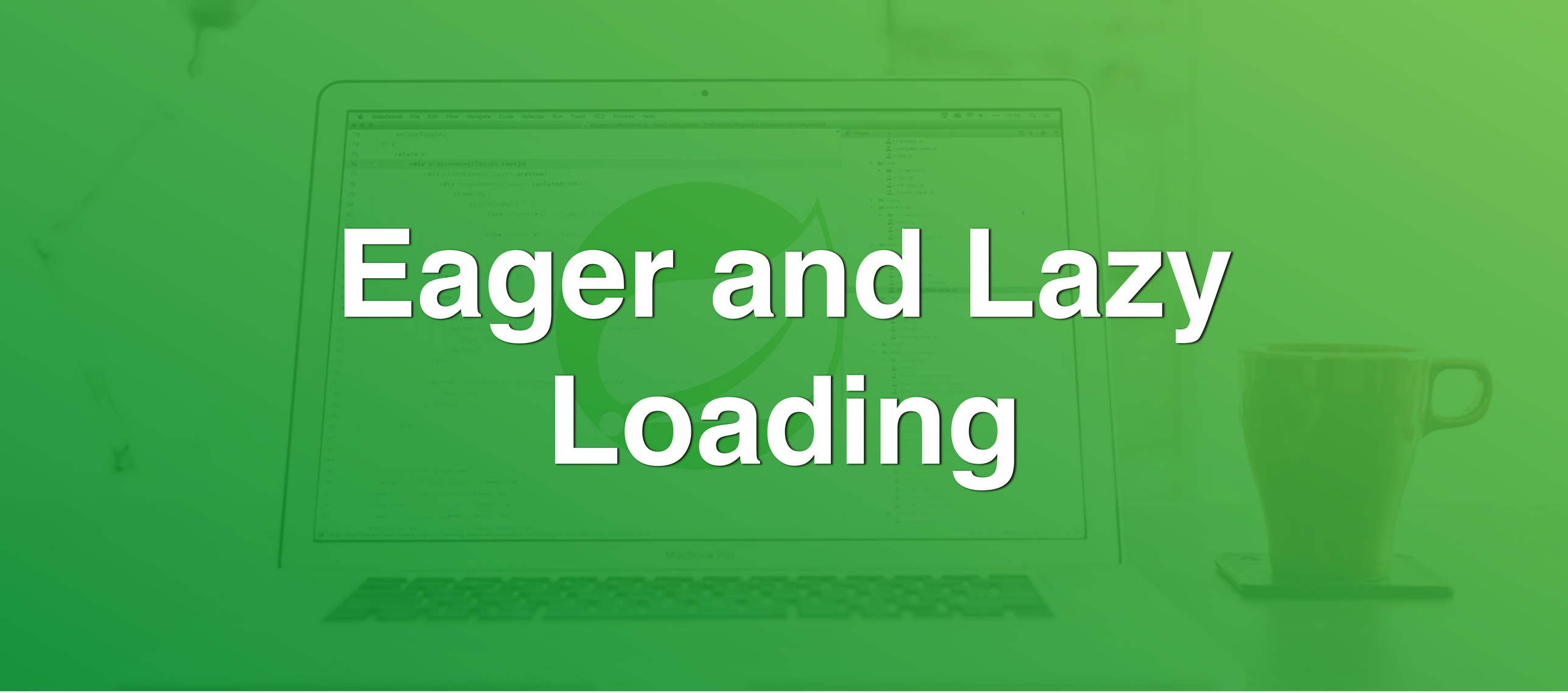
Instructor

Akın Kaldıroğlu

Expert for Agile Software Development and Java

Topics

- **Eager and Lazy Loading**
  - @Lazy
- **Ordering Beans**
  - @Order and @DependsOn
- **Bean Lifecycle**
  - Lifecycle Events
- **Spring IoC Container Lifecycle**



Eager and Lazy Loading

selsoft
build better, deliver faster

Eager and Lazy Loading - I

- As we know **Spring** loads all singleton beans eagerly in default.
- Prototype beans are loaded only when they are asked for from the context.
- **`default-lazy-init`** attribute of **`</beans>`** is used to declare that all singletons should be loaded lazily by giving **`true`** value in XML file.
- Or **`lazy-init`** attribute of **`</bean>`** can be used to do the same thing per bean basis.

Eager and Lazy Loading - II

- During bootstrap the container scans all given XML files and create instances of all beans declared as eager.
- If a bean is declared prototype it is not created until asked for from the context even though it is declared to be eagerly loaded.
  - Injecting a bean into another bean also causes creation of the bean instance.
- Declaring all singleton beans to be loaded eagerly may cause the bootstrap of the application to take some time but all bean instances would be ready to use when the application finishes starting.

@Lazy

@Lazy - I

- **Spring**'s default behavior to load singleton beans eagerly is also valid for both annotation-based declaration using `@Component` and for Java-based configuration using `@Bean` and `@Configuration`.
- For these two kinds of configuration `@Lazy` is used to specify that bsingleton eans should be loaded lazily.
- `org.springframework.context.annotation.Lazy` is an annotation that can be used both at class and method level.
- `@Lazy` has an attribute `value` which is `true` in default.

@Lazy - II

- When used with **`@Component`** and **`@Bean`**, **`@Lazy`** specifies that the singleton bean should be loaded lazily.
- When used with **`@Configuration`** **`@Lazy`** specifies that all of the singletons that will be created in methods with **`@Bean`** should be loaded lazily.
- Methods with **`@Bean`** can override what **`@Configuration`** specifies by declaring **`value`** attribute of **`@Lazy`** as **`false`**.
  - Using **`@Lazy`** with **`false`** in attribute **`value`** is very rare.

@Lazy - III

- `@Lazy` can also be used at injection points with `@Autowired` and `@Inject`.
- In this case the injected object's proxy is created and injected not real object itself.
- The creation of the real object is delayed until it is reached on the enclosing object.
- This mechanism is known as **proxy pattern** of GoF.

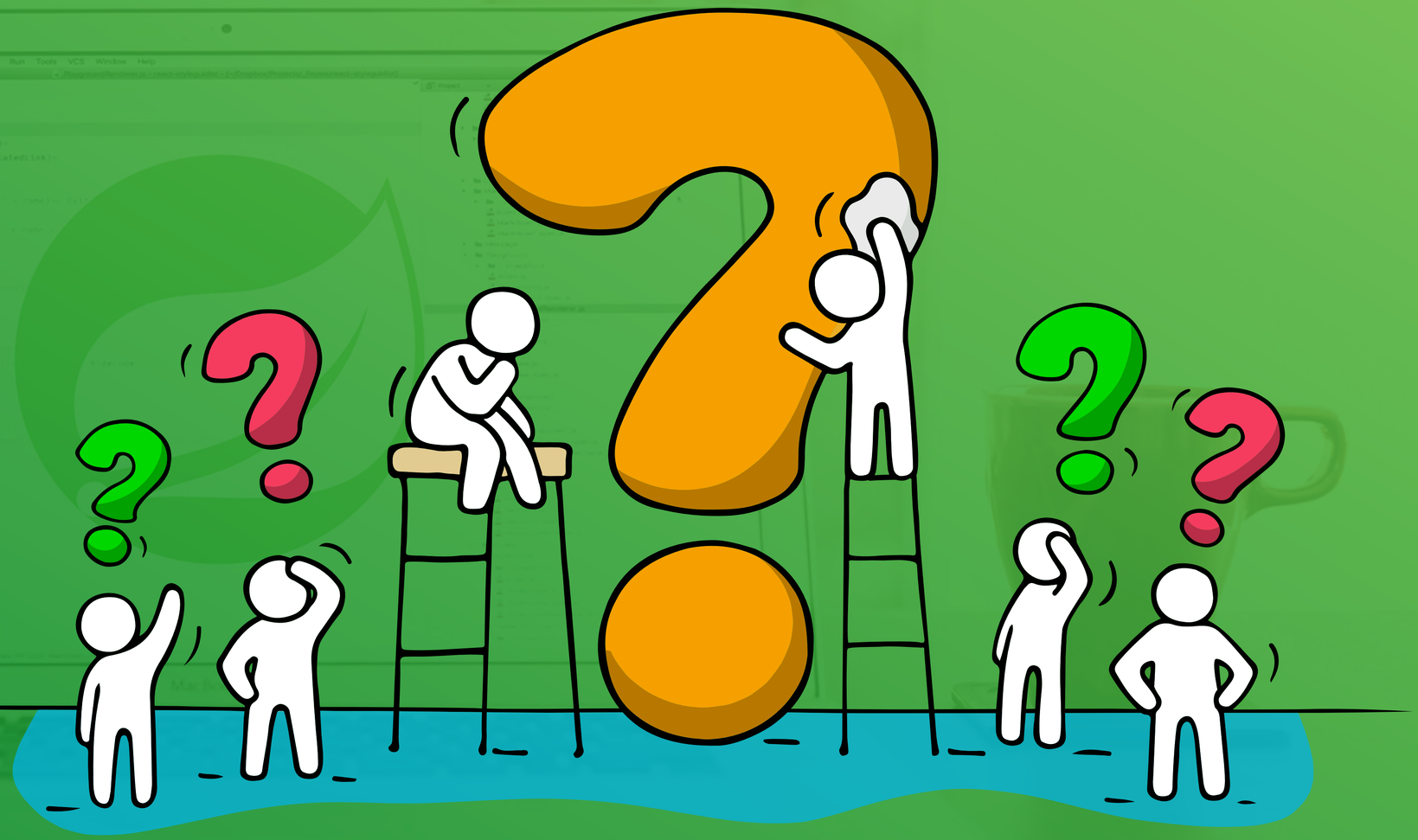
LazyExample

- `org.javaturk.spring.di.ch08.lazy.LazyExample`
  - `getBeans1()`
    - First comment out `@Lazy` in `BeanA` and don't retrieve beans from context.
    - Then use `@Lazy` in `BeanA` and don't retrieve beans from context.
  - `getBeans2()`
    - Run it with `@Lazy` in different places.

LazyExample

- **`org.javaturk.spring.di.ch08.lazy.LazyExample`**
  - **`getBeans1()`**
    - **`Mark`** both **`BeanA`** and **`BeanA`** **`@Lazy`**
    - Declare the point where **`BeanB`** is injected **`@Lazy`** in **`BeanA`**.
    - In **`getBeans1()`** retrieve **`BeanA`** but do not call its **`toString()`** method.
    - Observer whether **`BeanB`** instance is created.
    - What happens differently when **`toString()`** method of **`BeanA`** is called?

Time for Questions!



selsoft
build better, deliver faster

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

Ordering Beans

selsoft
build better, deliver faster

Order for Initialization and Injection

- There are mainly two kinds of order among beans:
  - Initialization order that is mainly determined by dependency relations.
  - Injection order
- Initialization order can be controlled by `DependsOn` annotation.
- There are several different methods to control injection order among beans.

Initialization Order and @DependsOn

@DependsOn - I

- **`org.springframework.beans.factory.annotation.DependsOn`** to declare that a bean depends on another bean.
- **`@DependsOn`** is used to influence the initialization order of the beans.
- **`@DependsOn`** takes an argument **`value`** of type **`String`** array which are the names of the dependent beans.
- If a bean declares **`@DependsOn`** to depend on another bean, IoC guaranties that another bean is initialized before the bean that declares **`@DependsOn`**.

@DependsOn - II

- `@DependsOn` also influences the destruction order of the singleton beans.
- If a bean declares `@DependsOn` to depend on another bean then the destruction of the declaring bean would happen before the depended bean.

@DependsOn - III

- `@DependsOn` doesn't have any effect at the class level if component-scanning isn't used.
- Depends on relationship can also be declared in XML file using `depends-on` attribute of `</bean>`.
- If a bean is declared with depends on relationship both in XML and with `DependsOn` annotation then `DependsOn` annotation is ignored and XML declaration is in effect instead.

@DependsOn - IV

- Be careful not to create circular dependencies!

DependsOnExample

- `org.javaturk.spring.di.ch08.dependsOn.DependsOnExample`
  - Run it without and with `@DependsOn` on `ABean`.

Injection Order

Order of Injection

- Injection order can be controlled by several mechanisms in **Spring**:
  - `@Order`
  - `@Priority` of JSR-250
  - `Ordered` interface
- Injection order does not influence initialization order which is mainly determined by dependency and `@DependsOn` relationships.

@Order - I

- `org.springframework.core.annotation.Order` is an annotation to control injection order of beans.
- `@Order` is used to sort beans for example in a list using a specific order.
- `@Order` has only one attribute `value` which is an `int`.
  - Its default value is `org.springframework.core.Ordered.LOWEST_PRECEDENCE`
- Beans with lower values in `@Order` are injected before than beans with higher values in `@Order`.

@Order - II

- Beans with the same order are injected with an arbitrary order.
- `order.org.springframework.core.OrderComparator` is responsible for this.

Ordered Interface

- `order.org.springframework.core.Ordered` is an interface to control the injection order of the beans.
- It has only one method `getOrder()` which returns the order as an integer.
- Ordering of injection using this interface is the same as `@Order`.
- Beans that return lower values are injected before than beans that return higher values.

OrderExample

- `org.javaturk.spring.di.ch06.order.OrderExample`

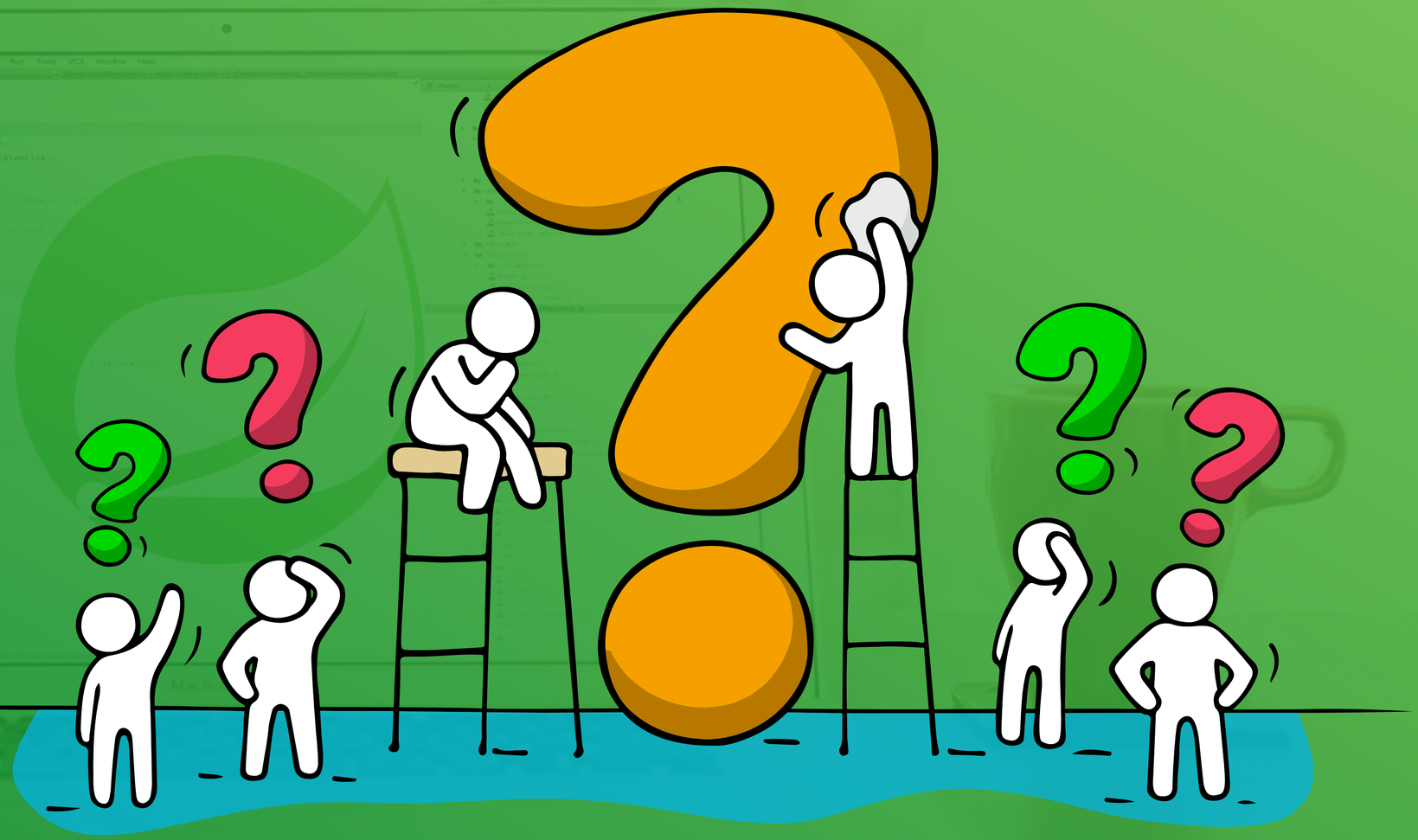
@Priority

- **`javax.annotation.Priority`** is an annotation that is part of JSR-250.
- **`@Priority`** is used to sort beans for example in a list using a specific order.
- **`@Priority`** has only one attribute **`value`** which is an **`int`**.
  - **`value`** is given generally a non-negative value.
- Beans with lower values are injected before than beans with higher values.

PriorityExample

- `org.javaturk.spring.di.ch06.priority.PriorityExample`

Time for Questions!



selsoft
build better, deliver faster

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

What is Bean Lifecycle?

selsoft

build better, deliver faster

What is Lifecycle? - I

- Lifecycle is life time that an object goes through.
- There are many milestones called events during the lifecycle of a bean.
- What lifecycle and events mean depend on environments.
  - For example in **Spring** events like creation of the bean instance, initialization of that instance with all its dependencies and destruction of the instance when its IoC contanier shuts down are all events in the lifecycle of the bean.
  - Or JPA has different events thus lifecycle model for entities.

What is Lifecycle? - II

- In non-managed environment all these events are created and managed by the programer.
- But in managed environments such as containers, lifecycle is managed eand events are fired by the container.
- Containers such as Java EE's servlet and EJB containers and **Spring** IoC container have their own models for the lifecycle of their obejcts such as servlets, EJBs and beans, respectively.
  - So they manage the lifecycle of its objects or beans and provide facilities such as notifications for those events.

Events and Callback Methods - I

- In managed environments when something important happens to the object whose lifecycle is managed by the container, the container creates an event and notifies those objects that would have an interest for that event.
- The method that receives notifications regarding the events are called **callback method**.
  - And objects that represent events are passed to callback methods.
- They are mostly methods of a specific interface and the container calls those methods and passes them event objects.

Events and Callback Methods - II

- And of course the objects of those interfaces should register themselves for those notifications i.e. events.
- This mechanism is implemented mostly using design patterns such as Observer of GoF (also called Publisher-Subscriber, Event-Notification etc.)

Bean Initialization

Bean Initialization Order - I

- As you might remember the bean initialization starts with a constructor call to create the bean instance.
- If there is a constructor with injection it must be only one and it is called to create the instance.
  - Of course all dependencies in the constructor must be satisfied.
- If there is no constructor with injection then there must be default constructor which is called to create the instance.
  - In this case dependencies might be on fields and properties.

Bean Initialization Order - II

- After the instance is created the IoC container looks for the dependencies in instance variables and properties that must be satistfied.
- First dependencies in instance variables are satisfied.
- Then dependencies in methods are satisfied.
- The physical declaration order determines the order of satisfaction of dependencies in fields and methods.

Limitations of XML Configuration

- For XML configuration only property and constructor injections are available.
  - There can be no declaration for the injection for fields and config methods in XML.
- But for annotation-based configuration `@Autowired` can be used with fields, constructors, property and config methods.

InitializationOrderExample

- `org.javaturk.spring.di.ch08.order.init.InitializationOrderExample`

Life-Cycle Events and Notifications

Post-Initialization & Pre-Destruction - I

- Two main notifications for beans are:
  - Post-initialization and
  - pre-destruction (or pre-destroy)
- Initialization of a bean includes creation of its instance and injection of all of its dependencies so that the instance becomes ready to provide service.
- Destruction of a bean is the destruction of its instance so it can not serve anymore.

Post-Initialization

- The post-initialization notification that is called right after the bean is initialized with all its dependencies can be used to provide custom actions after the initialized bean instance is available.
- The custom actions can not be done in constructors due to the fact that some of the dependencies might be instance variables and properties and for full initialization they need to be injected too.
- The custom actions can be something like notifying some other beans regarding its readiness or setting up environment for further activities, etc.

Pre-Destruction

- The pre-destruction notification that is called right before the bean is destroyed by the container can be used to clean up resources the bean may hold.
- Beans may hold references to other objects or resources outside the JVM.
- It would be nice if a bean frees all the resources it holds especially if they are not in control of JVM such as files.

Post-Initialization & Pre-Destruction - II

- If beans are loaded eagerly which is the default case for singletons then even though the beans are not asked from the context they are initialized and post-initialization events are raised.
- But in case of lazy loading, the beans are only initialized when they are asked from the context and only that time post-initialization events are raised.

Post-Initialization & Pre-Destruction - III

- While the post-initialization event is called regardless of the scope of the bean, the pre-destruction event is called only on singleton beans, container doesn’t call this event for prototypes.

- Prototype beans can receive the pre-destruction events via a custom post processor such as **`DestructionAwareBeanPostProcessor`**.

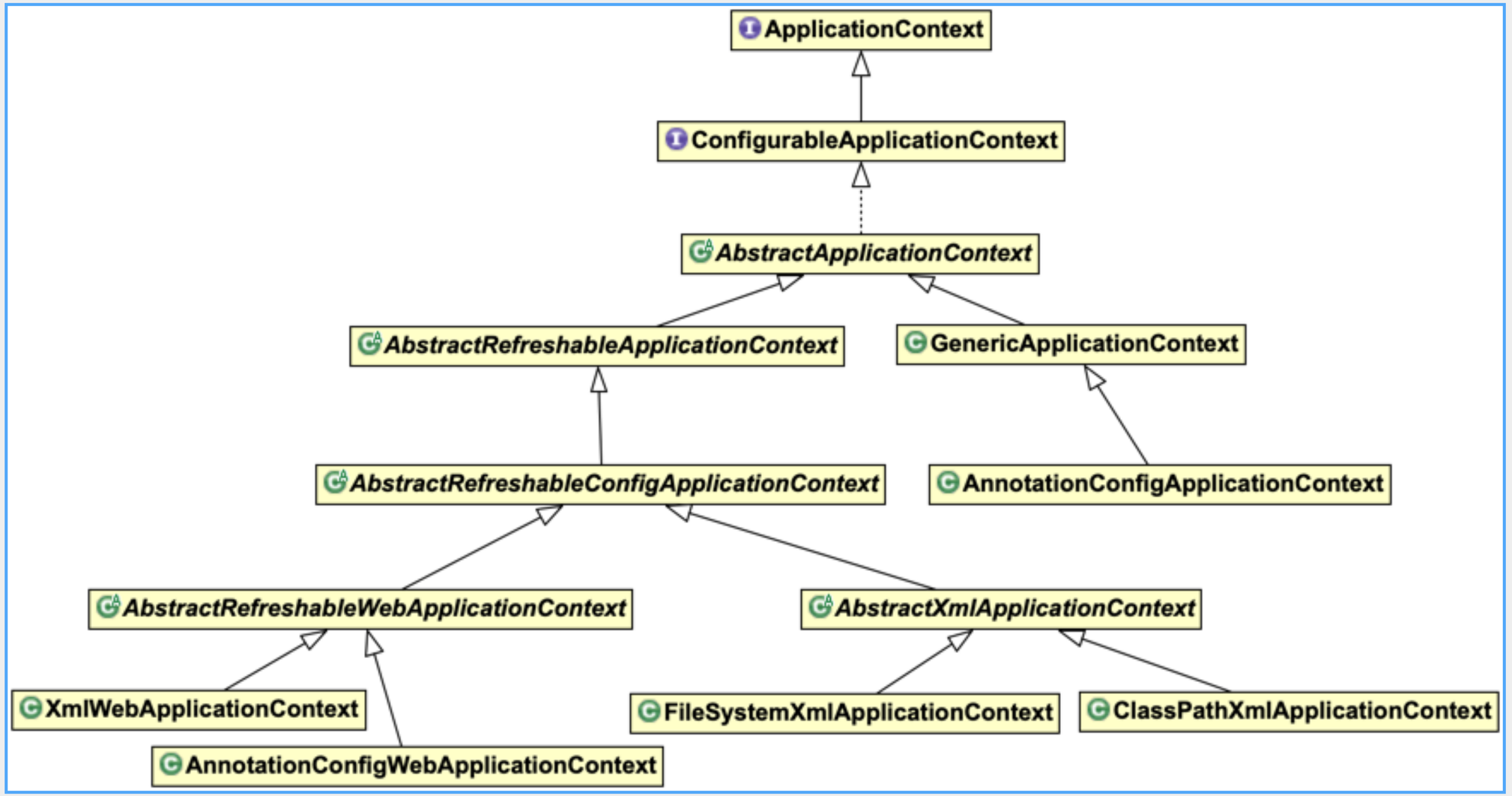
Lifecycle Events and Notifications

Lifecycle of ApplicationContext

Lifecycle of ApplicationContext

- `ApplicationContext` is the main object for bean configuration.
- `ConfigurableApplicationContext` is its sub-interface that adds configuration and lifecycle methods.
- Three main lifecycle methods are:
  - `refresh()`
  - `close()`
  - `registerShutdownHook()`

ApplicationContext Hierarchy



AbstractApplicationContext - I

- `AbstractApplicationContext` is the abstract base class for all context objects and implements all lifecycle methods:
  - `refresh()` either loads a new persistent representation of the configuration or refreshes new one discarding the old one.
  - `close()` closes the context, releasing all resources and locks that the implementation might hold.

AbstractApplicationContext - II

- **`registerShutdownHook()`** registers a shutdown hook named **`SpringContextShutdownHook`** with the JVM runtime, which is represented by **`java.lang.Runtime`** object.
- It closes the context on JVM shutdown unless it has already been closed before.
  - **`Runtime`** class has **`addShutdownHook(Thread hook)`** method that allows applications register a **`Thread`** object so that just before shutting down itself **`Runtime`** object executes them in **`java.lang.Shutdown`** class.
  - **`close()`** method removes any registered shutdown hook.

AbstractApplicationContext - III

- `AbstractApplicationContext` uses the **Template** design pattern.
- It has three abstract methods and even though it determines the main algorithm of the lifecycle of bean factory, it leaves the implementation of these three methods to concrete sub-classes.
- Those abstract methods that manage bean factory are:
  - `ConfigurableListableBeanFactory getBeanFactory()`
  - `void closeBeanFactory()`
  - `void refreshBeanFactory()`

AbstractApplicationContext - IV

- Even though it has an internal bean factory it leaves to its sub-classes issues regarding the bean factory such as how it is created, closed, refreshed, and where the storage for bean definitions should exists.
- And it manages all event and notification mechanisms by automatically registering **`BeanFactoryPostProcessor`**, **`BeanPostProcessor`**, and **`ApplicationListener`** which are defined as beans in the context.
- **`BeanFactoryPostProcessors`**, **`BeanPostProcessors`**, and **`ApplicationListeners`** will be explained soon.

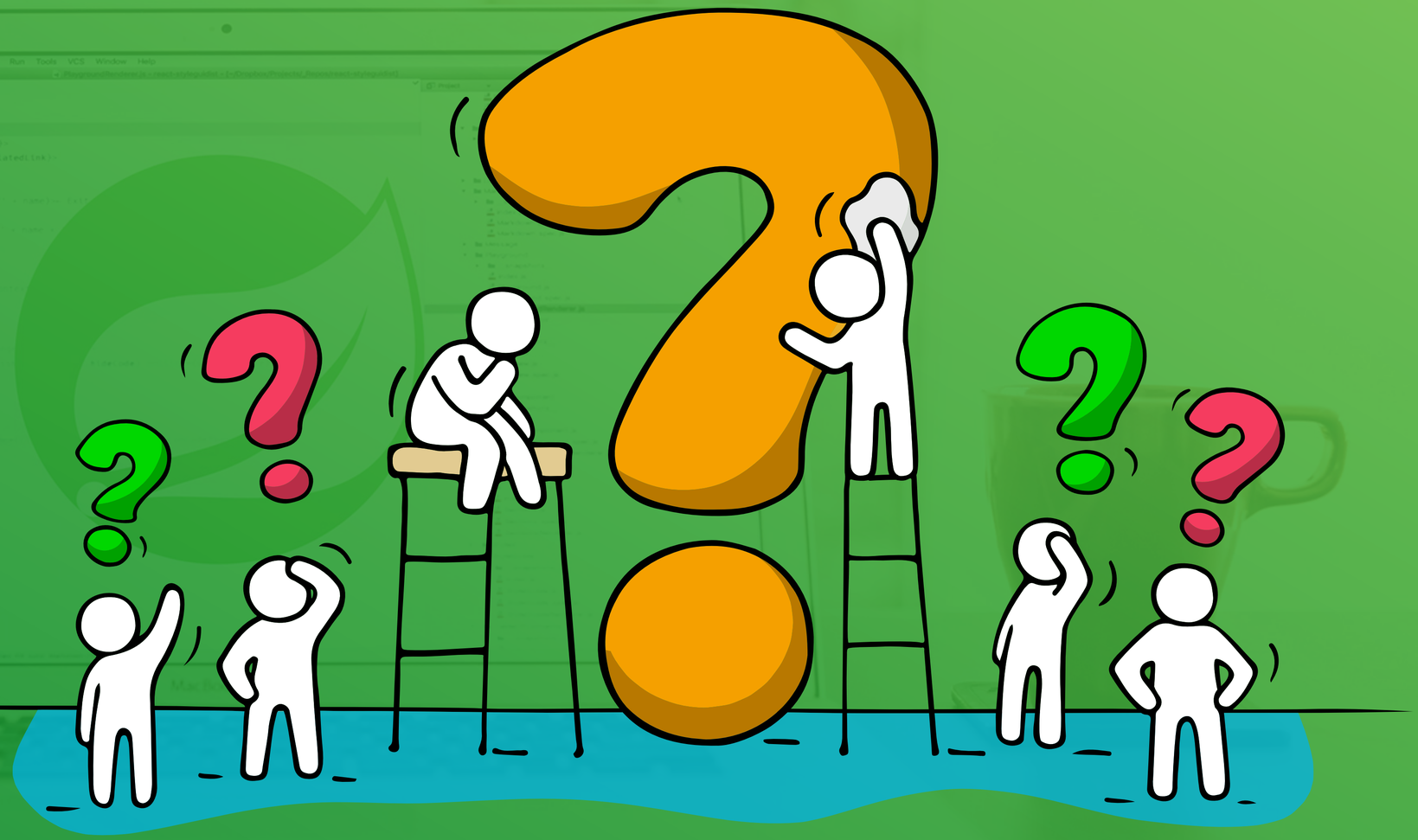
AbstractApplicationContext - V

- Have a look at the source code of following methods, which are template methods, in **`AbstractApplicationContext`**:
  - **`refresh()`**: Makes calls to abstract **`refreshBeanFactory()`** and **`getBeanFactory()`** methods.
  - **`close()`**: Makes call to abstract **`closeBeanFactory()`** method.
  - **`registerShutdownHook()`**: Makes call to abstract **`closeBeanFactory()`** method.

AbstractRefreshableApplicationContext

- `AbstractRefreshableApplicationContext` is the abstract base class that allows multiple `refresh()` calls.
- It creates a new bean factory instance for each `refresh()` call.
  - Typically a new `refresh()` call is necessary to load new bean definitions from new configuration locations.
- It has only one abstract method to load bean definitions:
  - `loadBeanDefinitions(DefaultListableBeanFactory beanFactory)`

Time for Questions!



selsoft
build better, deliver faster

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

Lifecycle Events and Notifications

Lifecycle of ApplicationContext

Lifecycle Methods: refresh()

refresh() - I

- **`refresh()`** is a startup method for the context objects.
- Base class for refreshable contexts is **`AbstractRefreshableApplicationContext`**.
- Calling its **`refresh()`** method creates a new internal bean factory instance.
- It loads or refreshes the persistent representation of the configuration, which might be from Java-based configuration, an XML file, a properties file, a relational database schema, or some other format.

refresh() - II

- So anytime an **`ApplicationContext`** object is initialized or refreshed all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the **`ApplicationContext`** object is ready for use.
- When a context is refreshed its callback is also called.
- All post-initialization callback methods on initialized beans are called.

refresh() - III

- If the configuration is passed to the constructors of **`ApplicationContext`** objects they scan all of the beans in configuration(s) and initialize all eagerly-loaded singletons making them ready for service.
  - Subclasses of **`ApplicationContext`** such as **`FileSystemXmlApplicationContext`** and **`ClassPathXmlApplicationContext`** automatically refresh themselves in their constructors when they load bean definitions.
  - Only after refresh it initializes and serves lazy-loaded beans upon request.

refresh() - IV

```
ApplicationContext context = new ClassPathXmlApplicationContext(“org/javaturk/spring/beans.xml”);
ApplicationContext context = new FileSystemXmlApplicationContext(“file:/Users/akin/beans.xml”);
```

refresh() - V

- If the context object does not receive configuration through its constructor then it needs a refresh to make all beans ready when a new configuration object is passed.

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext();
context.setConfigLocation("org/javaturk/spring/beans.xml");
context.refresh();
```

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();
```

- If a bean defined in a new configuration is asked for before refreshing the context the container throws **`IllegalStateException`** with a message **`org.springframework.context.annotation.AnnotationConfigApplicationContext@28ba21f3 has not been refreshed yet`**

refresh() - VI

- Some subclasses of **`ApplicationContext`**, in fact all subclasses of **`AbstractRefreshableApplicationContext`** such as **`FileSystemXmlApplicationContext`** and **`ClassPathXmlApplicationContext`** allow multiple refresh calls.
  - Spring calls them **hot refresh**.
- Calling **`refresh()`** method creates a new internal bean factory instance with a possible new configuration making all beans registered with the previous bean factory destroyed.

refresh() - VII

- After each new configuration set using **`setConfiguration()`** method **`refresh()`** should be called in order to load it.
- If **`refresh()`** is not called new configuration does not have any effect.
  - Remember subclasses of **`AbstractRefreshableApplicationContext`** such as **`FileSystemXmlApplicationContext`** and **`ClassPathXmlApplicationContext`** call **`refresh()`** in their constructors.

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(“org/javaturk/spring/beans.xml”);
…
context.setConfigLocation(“org/javaturk/spring/newBeans.xml”);
context.refresh();
```

ApplicationContextExample

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.ApplicationContextExample`
  - `refreshExample()`
  - Notice that `ClassPathXmlApplicationContext` requires `refresh()` call to make new configuration ready for use.

refresh() - VIII

- All **`GenericApplicationContext`** classes such as **`AnnotationConfigApplicationContext`** don't support multiple refresh calls; they allow only one refresh call.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig1.class);
…
ctx.register(AppConfig2.class);
// IllegalStateException! Multiple refresh calls not allowed! It's been called in constructor.
// ctx.refresh();
```

- When multiple **`refresh()`** calls are made the container throws **`IllegalStateException`** with a message *GenericApplicationContext does not support multiple refresh attempts: just call 'refresh' once*

AnnotationConfigApplicationContext - I

- **`AnnotationConfigApplicationContext`** is a standalone application context that suports annotation based configuration.
- It accepts as input component classes with **`@Component`**, **`@Configuration`**, and JSR-330 compliant classes using **`javax.inject`** annotations.
  - Most of the time its constructor receives a class that has **`@ComponentScan`** to scan classes with **`@Component`** or **`@Import`** to point to classes with **`@Configuration`**.

AnnotationConfigApplicationContext - II

- `AnnotationConfigApplicationContext` also makes a refresh call in its constructor.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AComponent.class);
```

```
@Component
public class AComponent{
  …
}
```

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
```

```
@Configuration
public class AppConfig{
  @Bean()
  public BeanA beanA(){
     return new BeanA();
  }
  …
}
```

```
@Import(AnotherConfig.class)
public class AppConfig{
  …
}
```

```
@ComponentScan("…")
public class AppConfig{
  …
}
```

```
@ComponentScan("…")
@Import(AnotherConfig.class)
public class AppConfig{
  …
}
```

AnnotationConfigApplicationContext - III

- **`AnnotationConfigApplicationContext`** also allows for registering classes both one by one using **`register(Class...)`** and with classpath scanning using **`scan(String…)`**.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(BeanA.class);
ctx.refresh();
```

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.scan("org.javaturk.spring.di.ch08.domain");
ctx.refresh();
```

- **`AnnotationConfigApplicationContext`** loads and makes ready new configuration upon **`refresh()`** call since default constructor does not call **`refresh()`**.

AnnotationConfigApplicationContext - IV

- As a sub-class of **`GenericApplicationContext`**, **`AnnotationConfigApplicationContext`** doesn't support multiple refresh; it allows only one refresh call in its lifetime.
- So single refresh call should be made either in its constructor if it loads a configuration or after loading new configuration using either **`register()`** or **`scan()`**.
- Once a refresh call is made there is no need for refresh after subsequent **`register()`** or **`scan()`** calls since it automatically loads and makes new configuration ready.

AnnotationConfigApplicationContext - V

- There is an exception to this:
  - If a class with `@Configuration` is registered using `register()` or `scan()` it always needs a new `refresh()` call to get effective.

AnnotationConfigApplicationContext - VI

- If the same bean with `@Component` is registered in a new configuration later registration overrides those in earlier classes.
- Same thing applies to `@Configuration` classes, `@Bean` methods in later classes will override those in earlier classes.
- Using `register()` or `scan()` is useful to build the configuration programmatically at run-time.

AnnotationConfigApplicationContextExample1

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.AnnotationConfigApplicationContextExample1`

AnnotationConfigApplicationContextExample2

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.AnnotationConfigApplicationContextExample2`

Lifecycle Events and Notifications

Lifecycle of ApplicationContext

Lifecycle Methods: close()

close() - I

- `close()` closes the application context.
- It also releases all resources and locks that the implementation might hold.
  - This includes destroying all cached singleton beans.
- When a context is closed its pre-destruction (or pre-destroy) callback is also called on singletons.

close() - II

- Once the context is closed, it reaches its end of life and cannot be refreshed, restarted or it doesn't serve any bean anymore.
- For example trying to read an object from a closed context causes **`java.lang.IllegalStateException`** with a message *BeanFactory not initialized or already closed…*
- This method can be called multiple times without side effects, subsequent **`close()`** calls on an already closed context will be ignored.

close() - III

- Although IoC container calls post-contruct notifications after refreshing the configuration on all beans it does not call pre-destroy notifications on prototypes.
- When the context is closed all beans, singleton or prototype, are destroyed so that the context is not available to serve them anymore.
  - What is different is about calling lifecycle destruction methods on prototypes, they are destroyed but don’t get notification automatically.

close() - IV

- The client must clean up prototype beans and release expensive resources that the prototype beans hold.
- Prototype beans can receive the pre-destruction events via a custom post processor such as **`DestructionAwareBeanPostProcessor`**.
- **Spring** says: In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java **`new`** operator. All lifecycle management past that point must be handled by the client.

Removing Beans from Context

- Beans can be manually removed from the context by calling **`removeBeanDefinition()`** method on **`org.springframework.beans.factory.support.BeanDefinitionsRegistry`**.

- When the bean instance is removed from **`BeanDefinitionRegistry`** of the context manually then it won't be available anymore from the context.

Destroying Beans

- When an **`ApplicationContext`** object is closed all beans it created continue to reside in the memory of JVM as initialized but the context is not available to serve any object or throw event anymore.
- When an **`ApplicationContext`** object is refreshed in its lifetime all beans it created before the refresh continue to reside in the memory as initialized but the context does not contain and manage them anymore.
- After the refresh **`ApplicationContext`** object starts serving new beans registered with new configuration.

ApplicationContextExample

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.ApplicationContextExample`
  - `closeExample()`
  - `removeBeansExample()`
  - Make scope of the beans `prototype` and see if there is any difference in terms of reaching them.

Lifecycle Events and Notifications

Lifecycle of ApplicationContext

Lifecycle Methods:

registerShutdownHook()

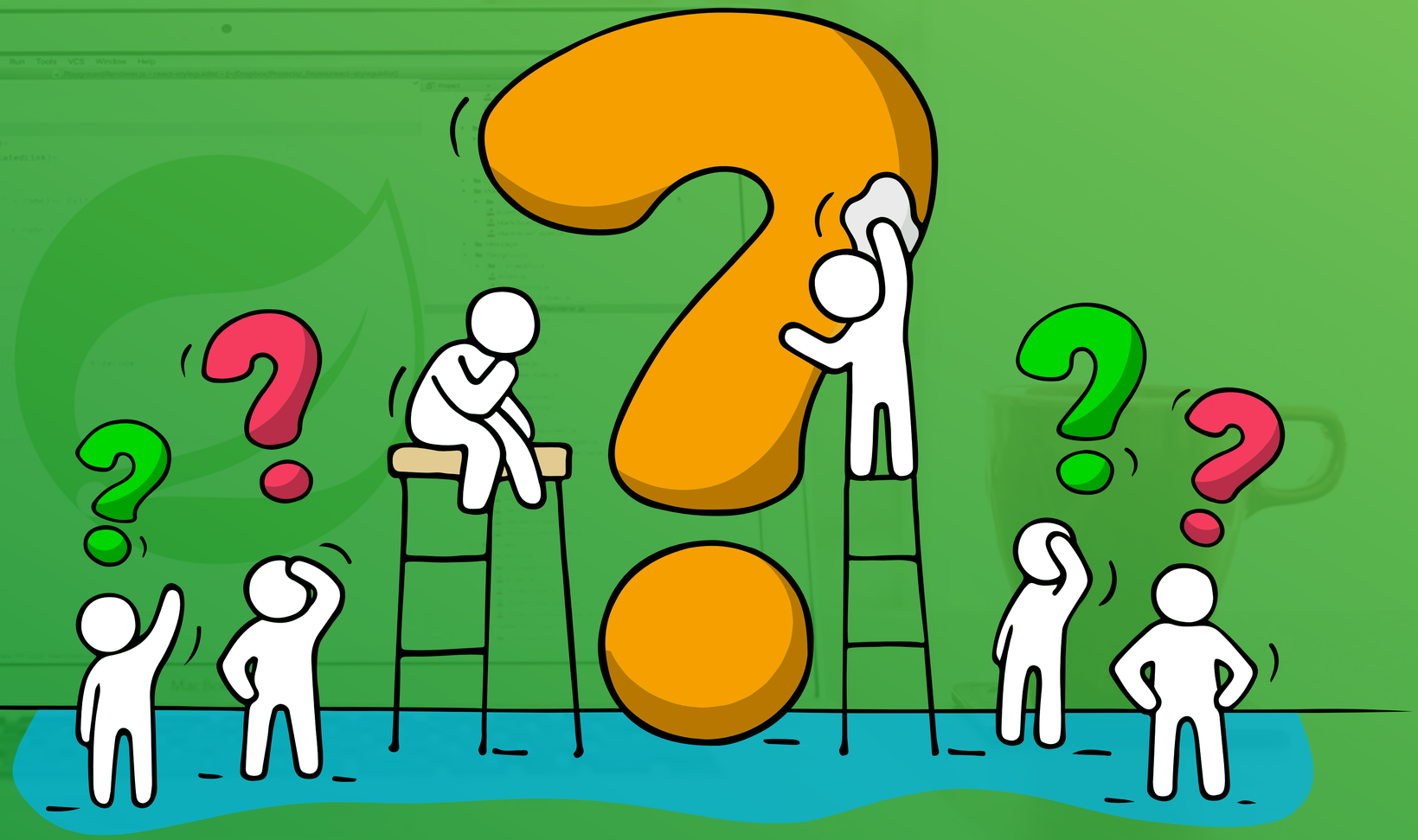
registerShutdownHook()

- Calling `registerShutdownHook()` registers a shutdown hook with the JVM.
- Purpose of calling this method is to ensure that a graceful shutdown will be done just before JVM exists.
- When JVM shuts down the context is also closed if it hasn’t been closed before.
- It calls the relevant destroy methods on singleton beans so that they can release their resources.

ApplicationContextExample

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.ApplicationContextExample`
  - `registerShutdownHookExample()`

Time for Questions!



selsoft
build better, deliver faster

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

Life-Cycle Events and Notifications

Lifecycle Callback Methods

Callback Methods

- Lifecycle callback methods defined for beans are called by IoC container automatically.
- Calback methods receive notifications regarding the events i.e. post-construct and pre-destroy events.
  - Of course only singletons receive pre-destroy events automatically.

Defining Methods for Events

- There are mainly three ways to let beans notified by the container about events:
  - **Methods**: Methods can be specified either in XML file or using JSR-250 annotations, `@PostConstruct` and `@PreDestroy`.
    - Spring recommends using `@PostConstruct` and `@PreDestroy`.
  - **Interfaces**: Beans implement specific interfaces of **Spring**.
  - **Annotations**: `@Bean` annotation has attributes to define methods to receive notifications.

Lifecycle Events and Notifications

Lifecycle Callback Methods

Methods

Life-Cycle Events in XML - I

- Lifecycle events can be configured using attributes of **`</bean>`** in XML:
  - **`init-method`**: For initialization event
  - **`destroy-method`**: For destroy event
- Events for all beans can be configured using attribute of **`</beans>`** :
  - **`default-init-method`** and **`default-destroy-method`**
  - These methods are only in affect when they are not overridden in **`</bean>`**.

Life-Cycle Events in XML - II

- Methods given for these attributes don’t receive any paramater but can have a return type.
- These methods can have any access modifier.

PostConstruct & PreDestroy - I

- `PostConstruct` and `PreDestroy` in `java.annotation package` are two call back method annotations for lifecycle events of a bean.
- They are part of JSR-250.
- They can be applied to any method on the bean.
  - Methods can be `private`, don't take any parameter and can return values.
- Multiple lifecycle events can be configured for a bean.

PostConstruct & PreDestroy - II

- In order for the container to process these annotations either `<context:annotation-config />` or `<context:component-scan />` must be in XML file or `@ComponentScan` must be used.

Closable and AutoClosable

- For beans that implement `java.lang.AutoCloseable` or `java.io.Closeable` interfaces, `close()` method can be defined in XML for pre-destruction event.

Exception

- IoC throws **`org.springframework.beans.factory.support.BeanDefinitionValidationException`** when it can't find declared methods in **`</bean>`** in XML on bean.

- No exception is thrown if a default notification method declared in **`</beans>`** is not found on a bean.

MethodLifecycleExample1

- **`org.javaturk.spring.di.ch08.lifecycle.MethodLifecycleExample1`**
  - First configure beans with their own callback methods using </bean> in XML.
  - Then use **`</beans>`** for callback methods available all beans.
  - Configure beans as lazy-loaded and observe when initialization callbacks are called.

MethodLifecycleExample1

- Configure beans as prototype and observe if pre-destroy callbacks are called.
- In XML configuration enable classpath scanning by `<context:component-scan base-package="...">` to see the effect of `@PostConstruct` and `@PreDestroy`.

MethodLifecycleExample2

- `org.javaturk.spring.di.ch08.lifecycle.callback.method.MethodLifecycleExample2`
  - Configure beans as eager-loaded which is default and lazy-loaded and observe when initialization callbacks are called.
  - Configure beans as prototype and observe if pre-destroy callbacks are called.

Lifecycle Events and Notifications

Lifecycle Callback Methods

Interfaces

Spring's Life-Cycle Events

- Two interfaces in `org.springframework.beans.factory` to receive callback notifications are:
  - `InitializingBean`'s `afterPropertiesSet()` method is called after all properties set on the bean.
  - `DisposableBean`'s `destroy()` method is called on the destruction of the bean.
- **Spring** suggests using `@PostConstruct` and `@PreDestroy` not to couple the code to **Spring**.

Lifecycle - SmartLifecycle Interfaces

- `org.springframework.context.Lifecycle` is a plain contract for explicit start and stop notifications.
  - It is not meant for automatic notifications.
- Its sub-interface `org.springframework.context` `SmartLifecycle` receives callbacks for events automatically.
- `SmartLifecycle` provides asynchronous stop event too.

InterfaceLifecycleExample

- `org.javaturk.spring.di.ch08.lifecycle.callback.interfaces.InterfaceLifecycleExample`

greeting20

- `org.javaturk.spring.di.ch08.greeting.greeting20.Application`

Lifecycle Events and Notifications

Lifecycle Callback Methods

Annotations

Life-Cycle Events in @Bean

- Life-cycle events can be configured in `@Bean` using attributes:
  - `initMethod`: For initialization event
  - `destroyMethod`: For destroy event
- Methods given for these attributes don't receive any paramater but can have a return type.
- These methods can have any access modifier.

close() and shutdown() Methods

- `public close()` or `shutdown()` methods, either of them is automatically called with a destruction callback if beans defined with Java configuration using `@Bean` have them.

- To avoid these methods to be called as callback this feature can be turned off by overriding `destroyMethod` attribute of `@Bean` and giving it an empty `String` such as `@Bean(destroyMethod="")`.

AnnotationLifecycleExample

- `org.javaturk.spring.di.ch08.lifecycle.callback.annotation.AnnotationLifecycleExample`

Lifecycle Events and Notifications

Lifecycle Callback Methods

Which One?

Which One?

- Using interfaces makes beans dependent on them.
- Using JSR-250 annotations makes the code more portable than other ways.
- If portability is not a priority then using interfaces might be a better choice.
  - Having a standard methods for lifecycle callbacks is a good practice.

Order of Callback Methods - I

- If there are more than one callback method for events their order of call is:
  - Initialization methods are called in the order:
    - Methods annotated with **`@PostConstruct`**
    - **`afterPropertiesSet()`** of **`InitializingBean`** interface
    - A custom configured **`init()`** method

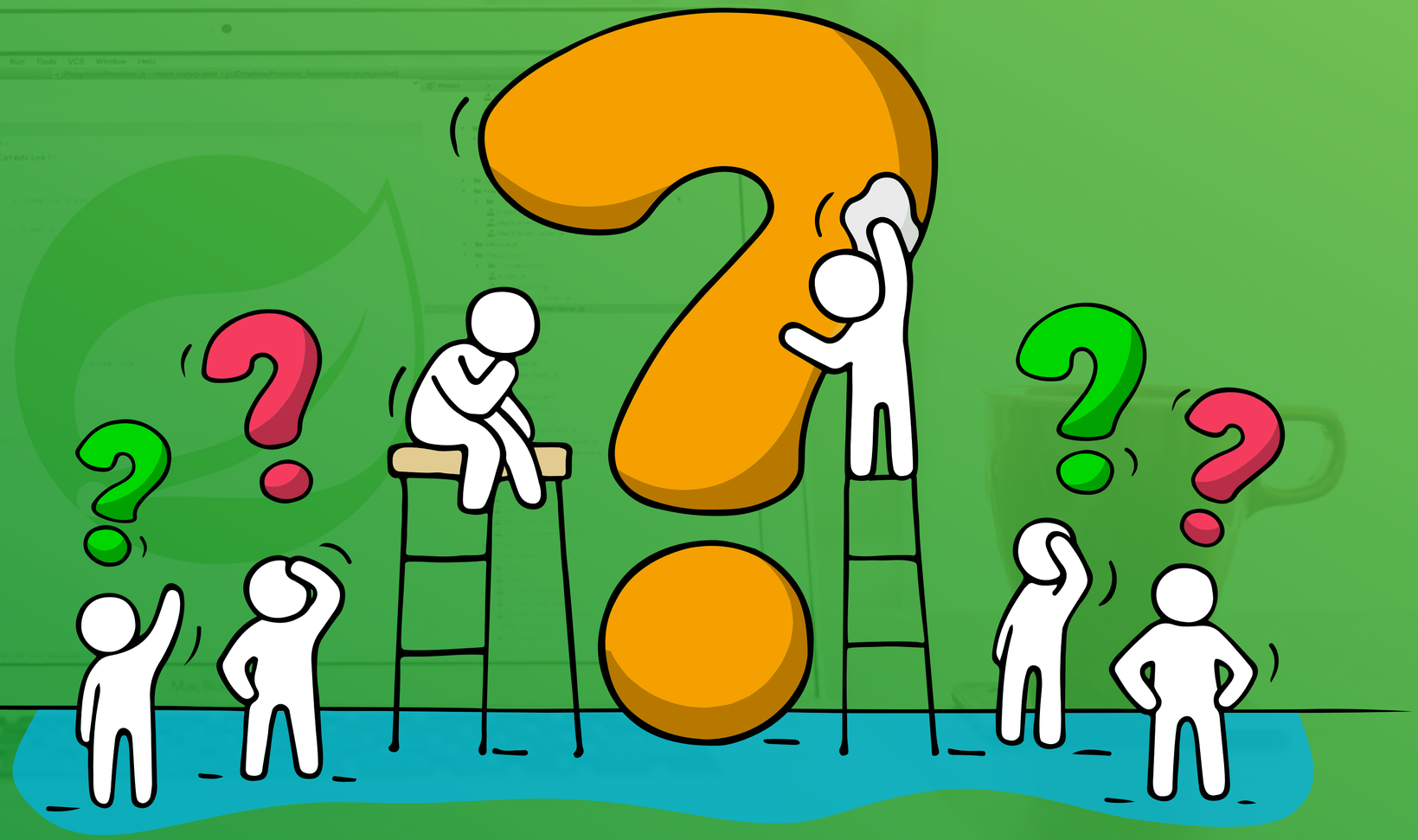
Order of Callback Methods - II

- Destroy methods are called in the same order:
  - Methods annotated with `@PreDestroy`
  - `destroy()` as defined by the `DisposableBean` callback interface
  - A custom configured `destroy()` method

AnnotationConfigApplicationContextExample

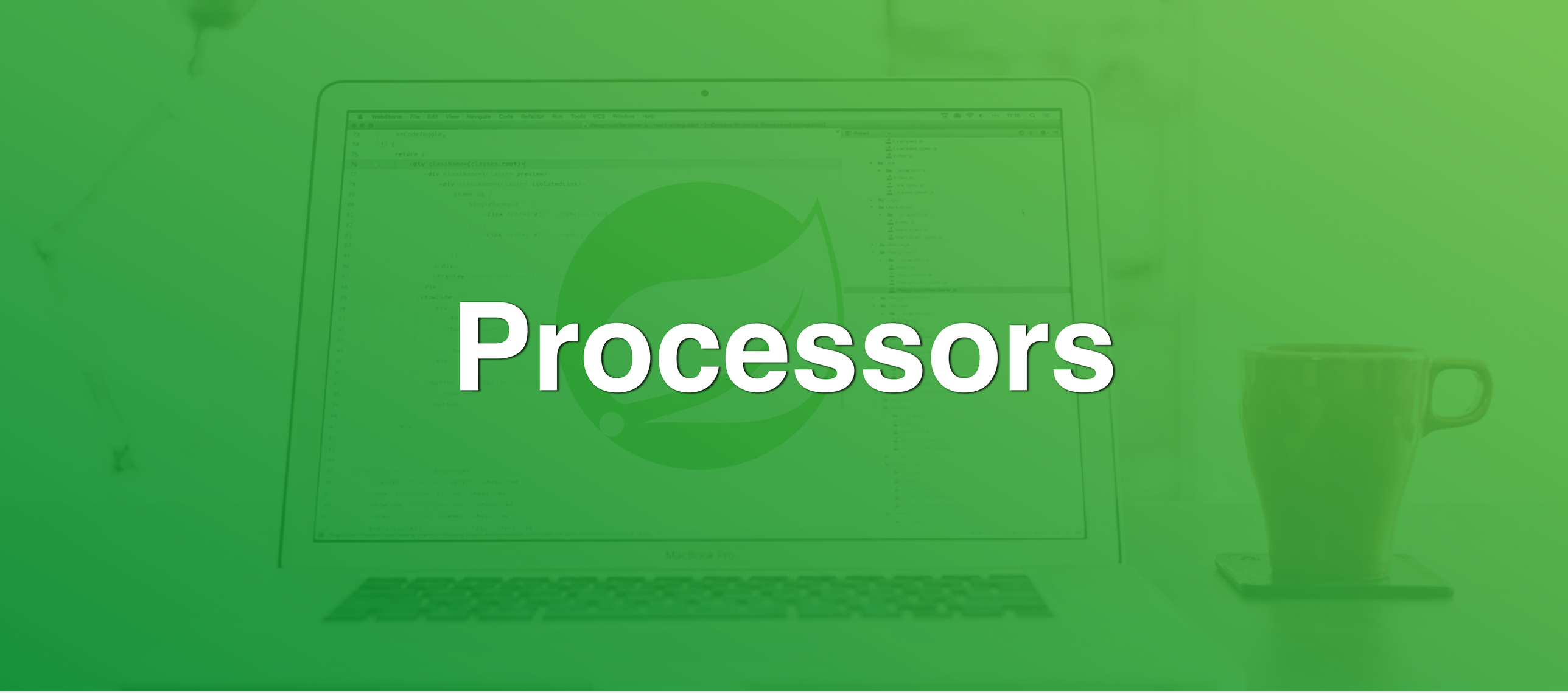
- `org.javaturk.spring.di.ch08.lifecycle.callback.example.AnnotationConfigApplicationContextExample`
- This is to understand how `register()` and `scan()` methods work with or without `refresh()`.
- Callback methods on the beans are defined.

Time for Questions!



selsoft
build better, deliver faster

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



Processors

selsoft
build better, deliver faster

Extensions to ApplicationContext

- IoC has several interfaces that can be used to extend `ApplicationContext` object.
- They are called **processor**.
- Two main processors are:
  - `BeanPostProcessor`
  - `BeanFactoryPostProcessor`
- They are in `org.springframework.beans.factory.config` package.

Processors

- **`BeanFactoryPostProcessor`**s run before **`BeanPostProcessor`**s.
- Processors are autodetected by **`ApplicationContext`** if they are in classpath.
- If they are created by a **`@Configuration`** class, **`@Bean`** methods that return their instances should be **`static`** to avoid the initialization of other beans in other **`@Bean`** methods of the same class.

BeanPostProcessor - I

- `org.springframework.beans.factory.config.BeanPostProcessor` allows custom modification for beans.
- It has two `default` methods that can be implemented by choice:
  - `default Object postProcessAfterInitialization(Object bean, String beanName)`
  - `default Object postProcessBeforeInitialization(Object bean, String beanName)`

BeanPostProcessor - II

- `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` are called just before and after any bean initialization callbacks such as a method with `@PostConstruct` annotation or `InitializingBean`'s `afterPropertiesSet()` method.
- When these methods are called the beans had already been created and populated with dependencies.

BeanPostProcessor - III

- Any custom logic such as programmatically setting some dependencies right after the beans instance are created and just before any initialization callbacks are called can be implemented in `postProcessBeforeInitialization()` method.

- Custom logic such as checking if all bean dependencies are met or making some changes to bean or its dependencies after the initialization can be implemented in `postProcessAfterInitialization()` method.

BeanPostProcessorExample

- `org.javaturk.spring.di.ch08.processor.beanPostProcessor.BeanPostProcessorExample`

BeanFactoryPostProcessor - I

- `org.springframework.beans.factory.config.BeanFactoryPostProcessor` allows custom modification for bean factory.
- It has one abstract method:
  - `void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)`
- It can be used to get information for example about all registered beans or override some of the properties of bean factory.

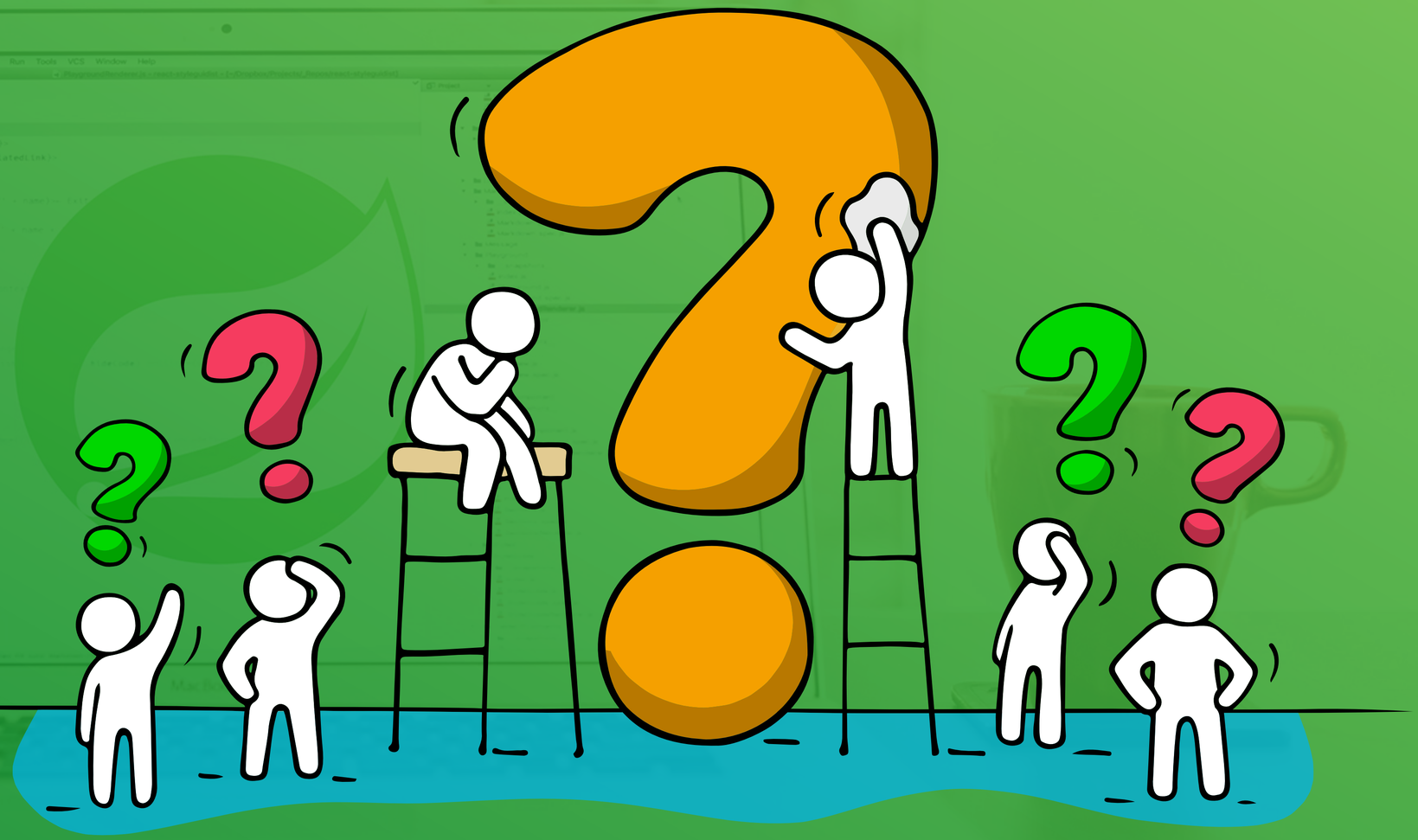
BeanFactoryPostProcessor - II

- Notice that when `postProcessBeanFactory()` is called no registered bean has not been initialized yet.
- So `postProcessBeanFactory()` should be used to modify bean definitions not bean instances because of the fact that it is called before any bean initialization and trying to initialize a bean inside it may result in premature bean instances.

BeanFactoryPostProcessorExample

- `org.javaturk.spring.di.ch08.processor.beanFactoryPostProcessor.BeanFactoryPostProcessorExample`
  - First run `getBeans1()` then `getBeans2()` and compare results.
    - Try to enforce bean initialization in `postProcessBeanFactory()` and see what happens.

Time for Questions!



selsoft
build better, deliver faster

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

Awareness Interfaces

selsoft

build better, deliver faster

Aware Interfaces

- **Spring** has a number of aware interfaces that provide implementing beans with awareness related to different objects of IoC:
  - **`BeanNameAware`**: Lets the bean know about its name.
  - **`BeanFactoryAware`**: Lets the bean know about its bean factory.
  - **`ApplicationContextAware`**: Lets the bean know about its application context.
- Their order is **`BeanNameAware`**, **`BeanFactoryAware`** and then **`ApplicationContextAware`**.

Aware Interface

- `org.springframework.beans.factory.Aware` is their parent and a marker interface.
  - It has no functionality.
- It marks the implementer bean as an observer or listener and makes it eligible for notifications.
- `Aware` interfaces are called after the bean is initialized.

BeanNameAware

- `BeanNameAware` lets the bean know about its name.
- **Spring** does not recommend a bean to depend on its name.

BeanFactoryAware

- `BeanFactoryAware` lets the bean know about its bean factory.
- The bean may want to reach other beans created by the same bean factory.
- If a bean has a reference to another bean for example through a dependency there is no need to use this interface.
- It can be used to reach other beans that a bean does not have a direct access.

ApplicationContextAware

- `ApplicationContextAware` lets the bean know about its context.
- The bean may want to reach information that only `ApplicationContext` object can serve such as system properties, file resources, etc..

Warning

- **Spring** warns in its reference document against using these interfaces, it says:
  - Note again that using these interfaces ties your code to the **Spring** API and does not follow the Inversion of Control style.

AwareExample

- `org.javaturk.spring.di.ch08.aware.AwareExample`

Exercise

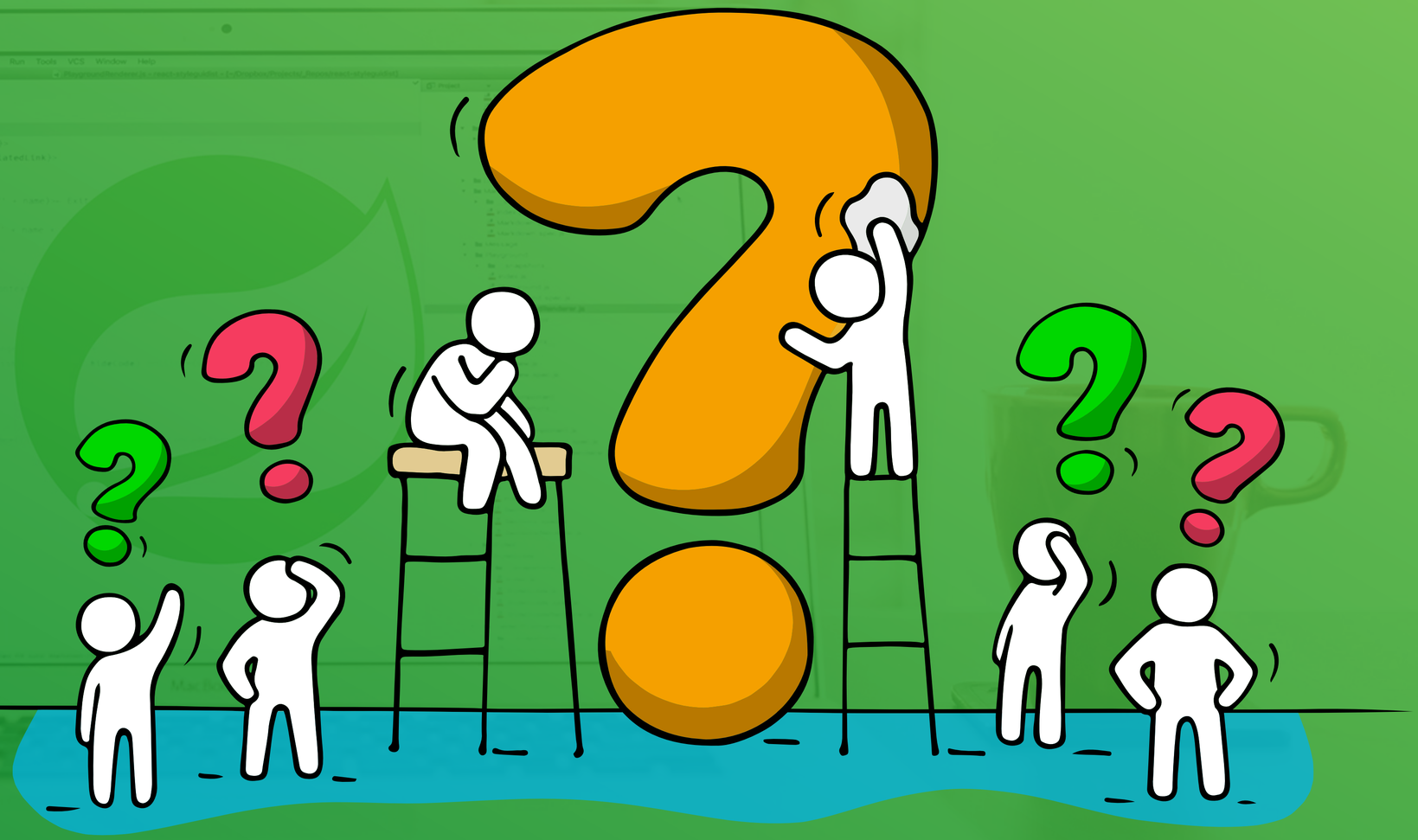
Exercise

- How to implement a solution to destroy prototype beans using **`BeanPostProcessor`** and **`BeanFactoryAware`**?
- Solution:
  - Create a singleton bean that implements **`Disposable`** interface.
  - In the **`destroy()`** method of **`Disposable`** interface it will call **`destroy()`** method of all prototype beans that it collects from **`BeanFactory`**.

Exercise

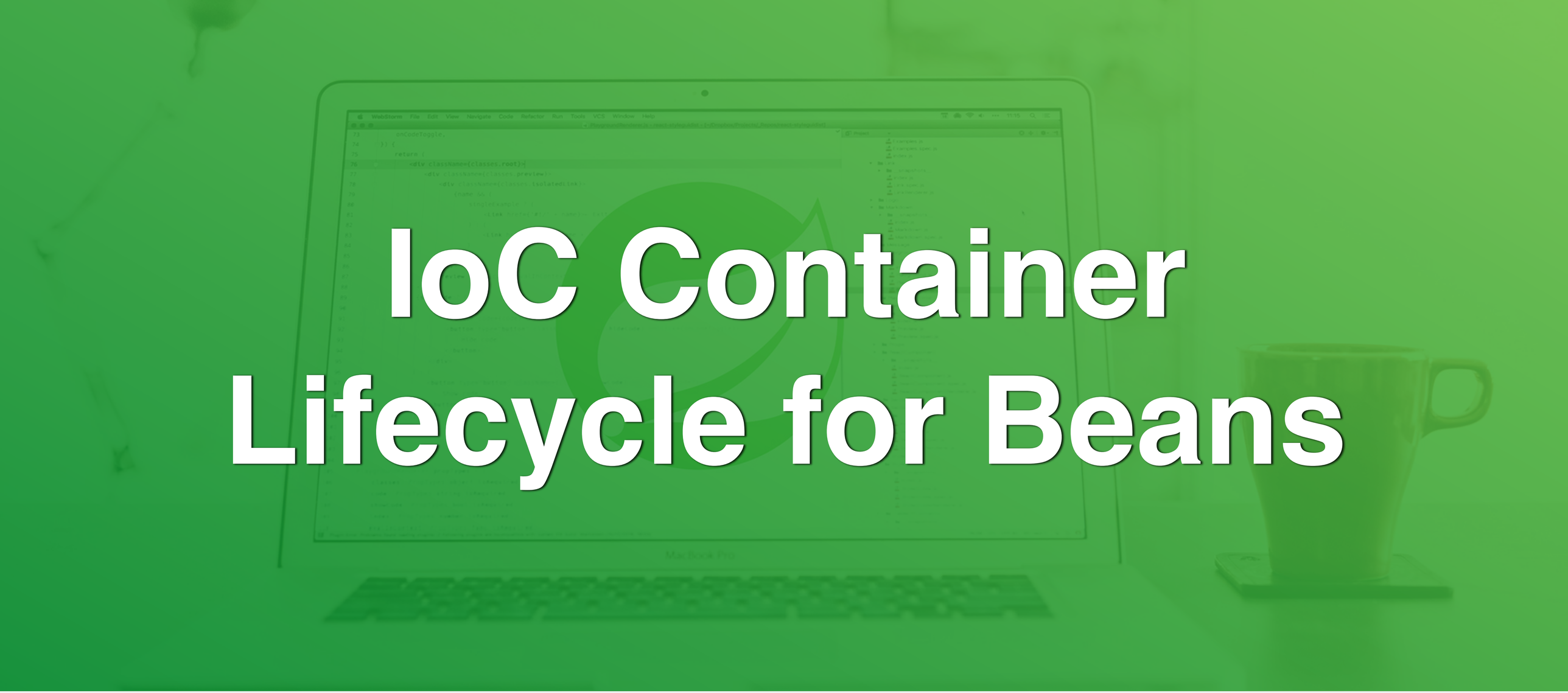
- `org.javaturk.spring.di.ch08.lifecycle.callback.prototype.PrototypeDestroyerExample`

Time for Questions!



selsoft
build better, deliver faster

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



IoC Container Lifecycle for Beans

selsoft
build better, deliver faster

IoC Lifecycle - I

- The steps of lifecycle that **Spring** IoC container goes through regarding beans:
  - It scans given XML files and classpath for beans
  - It creates beans using dependency relationships
  - It populates beans with their dependencies
  - It calls **`Aware`** interfaces
  - It calls **`postProcessBeforeInitialization()`** method of **`BeanPostProcessor`**

IoC Lifecycle - II

- It calls post-initialization callbacks
- It calls `postProcessAfterInitialization()` method of `BeanPostProcessor`

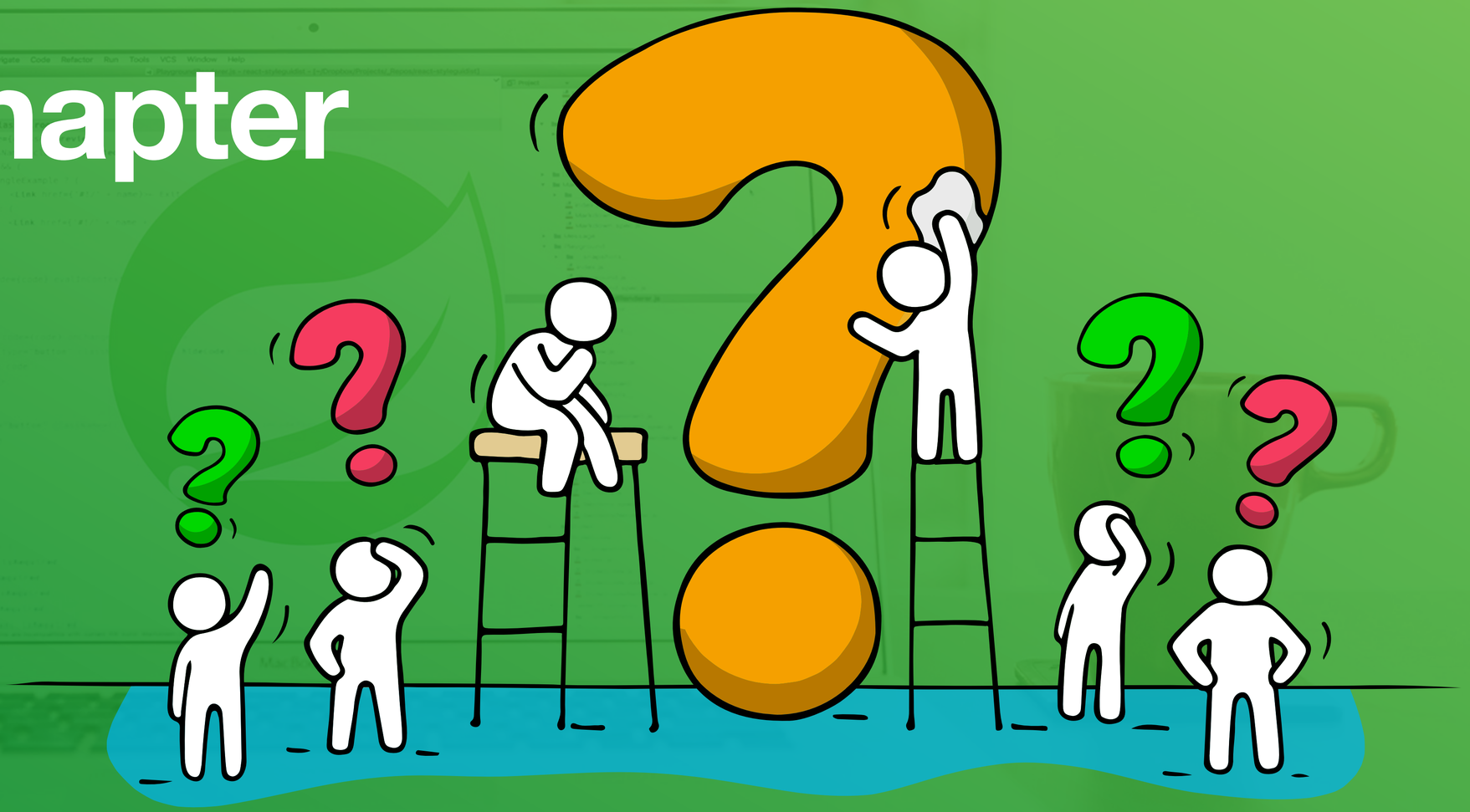
- Then bean is ready to use.
- And just before closing `ApplicationContext`
  - It calls pre-destruction callbacks

ApplicationContextLifecycleExample

- `org.javaturk.spring.di.ch08.lifecycle.applicationContext.lifecycle.ApplicationContextLifecycleExample`

End of Chapter

Time for Questions!



selsoft
build better, deliver faster

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