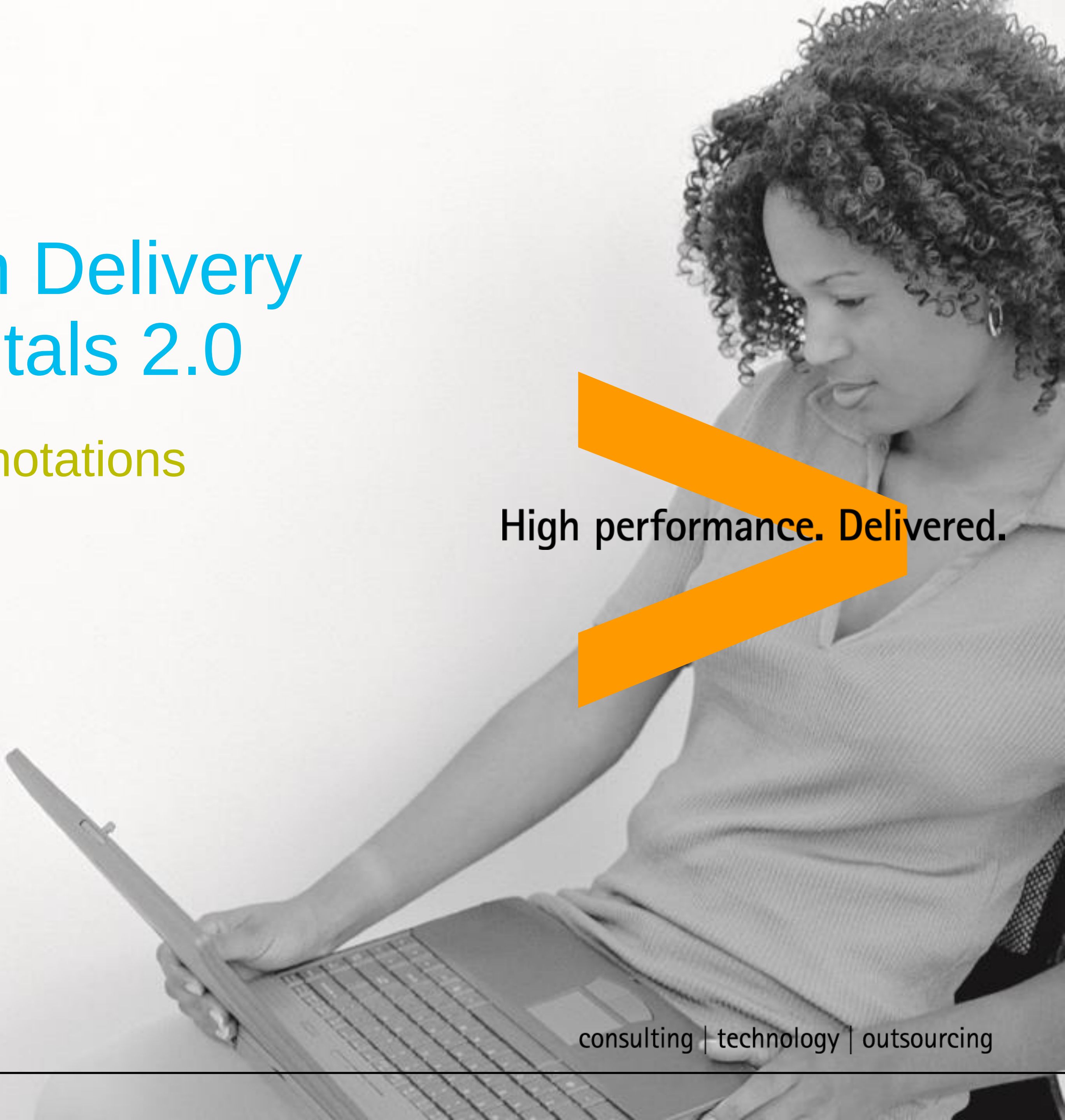
Application Delivery Fundamentals 2.0

Spring Core Annotations

High performance. Delivered.



consulting | technology | outsourcing

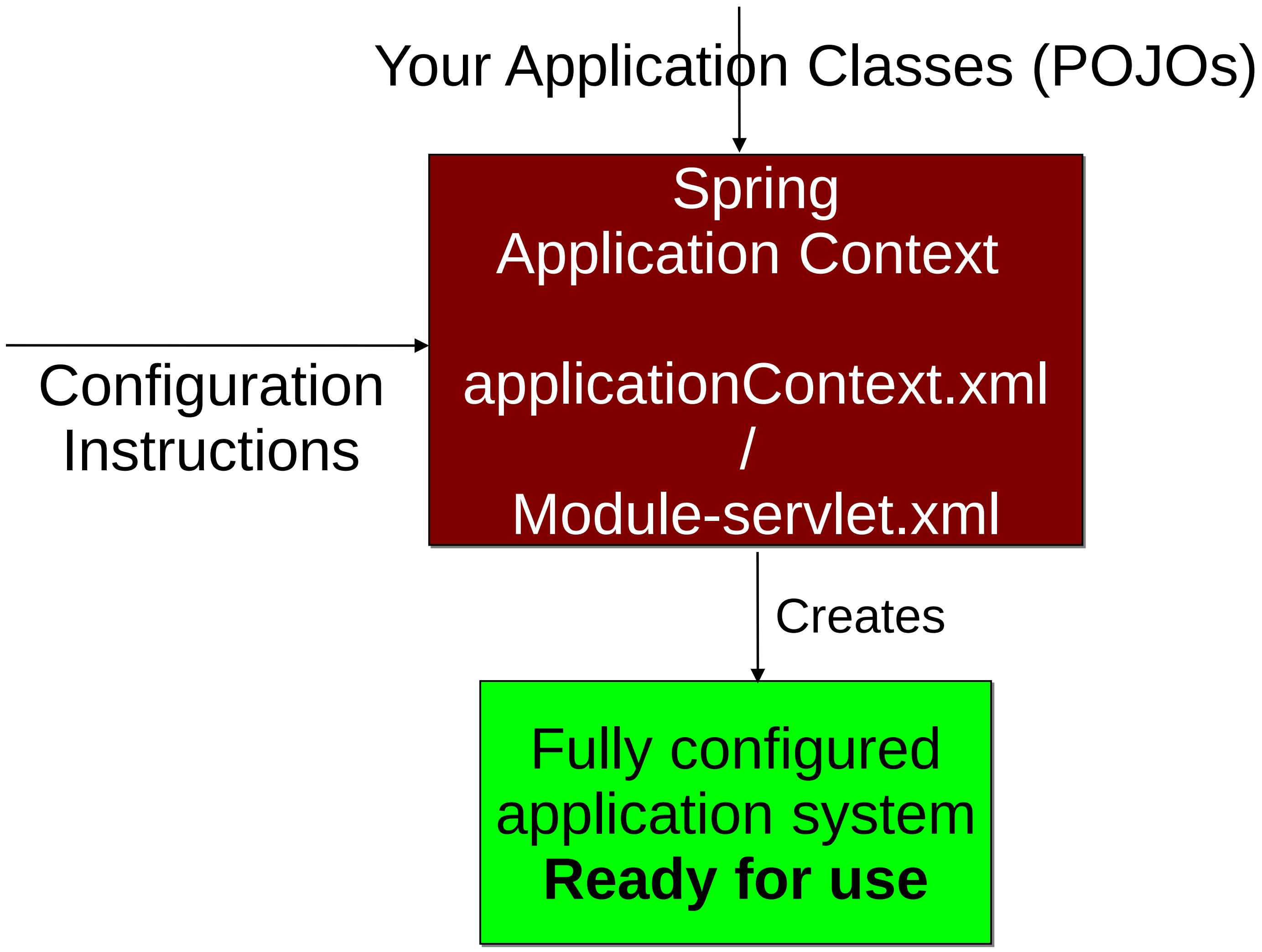
Course Goals / Objectives

- At the end of this module, participants will be able to:
  - Spring annotations
  - Annotation Configuration
    - @Autowired
    - @Component
    - @Qualiifier
  - Java Based Configuration Annotation
    - @Configuration
    - @Bean
  - Sample Code
  - Activity

Agenda

- This module will cover the following topics:
  - Some of the Spring Annotations
  - XML Configuration & Annotations
  - Java Based Configurations
  - When use what?
  - Summary

How Spring works


Your Application Classes (POJOs)
Configuration Instructions
Spring Application Context
applicationContext.xml / Module-servlet.xml
Creates
Fully configured application system **Ready for use**

Bean Injection

```
public class TransferServiceImpl implements

TransferService {

   // Constructor Injection
    public TransferServiceImpl(AccountRepository ar) {
            this.accountRepository = ar;
    }
   …

// OR – Setter Injection

AccountRepository accountRepository;

     public setAccountRepository (AccountRepository ar) {
          this.accountRepository = ar;
     }

}
```

Injecting AccountRepository Bean to TransferServiceImpl

Constructor Injection – XML Configuration

```
<beans>

   <bean id="transferService" class="app.impl.TransferServiceImpl">
      <constructor-arg ref="accountRepository" />
   </bean>

   <bean id="accountRepository" class="app.impl.JdbcAccountRepository">
      <constructor-arg ref="dataSource" />
   </bean>

   <bean id="dataSource" class="com.mysql.jdbc.Driver">
      <property name="URL" value="jdbc:mysql://localhost:3306/codingtondb" />
      <property name="user" value="root" />
      <property name="password" value="abcd1234" />
   </bean>

</beans>
```

Constructor Injection

Setter Injection – XML Configuration

```
<beans>
   <bean id=“transferService” class=“app.impl.TransferServiceImpl”>
      <property name=“accountRepository” ref=“accountRepository” />
   </bean>

   <bean id=“accountRepository” class=“app.impl.JdbcAccountRepository”>
      <property name=“dataSource” ref=“dataSource” />
   </bean>

   <bean id=“dataSource” class=“com.mysql.jdbc.Driver”>
      <property name=“URL” value=“jdbc:mysql://localhost:3306/codingtondb” />
      <property name=“user” value=“root” />
      <property name=“password” value=“abcd1234” />
   </bean>
</beans>
```

Setter Injection

- Place holder (Setter – Getter methods) for injecting bean in parent class.

@Autowired

```
•   public class TransferServiceImpl implements TransferService {
•       @Autowired
•       public TransferServiceImpl(AccountRepository ar) {
•           this.accountRepository = ar;
•       }
•       …
•   }
```

```
public class JdbcAccountRepository implements AccountRepository {
    @Autowired
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    …
}
```

@Autowired – XML Configuration

```
<beans>

   <bean id="transferService" class="app.impl.TransferServiceImpl" />

   <bean id="accountRepository" class="app.impl.JdbcAccountRepository" />


   <bean id="dataSource" class="com.mysql.jdbc.Driver">
      <property name="URL" value="jdbc:mysql://localhost:3306/codingtondb" />
      <property name="user" value="root" />
      <property name="password" value="abcd1234" />
   </bean>

<context:annotation-config/>

</beans>
```

No need to specify constructor-args / Setter reference

looks for annotations on beans only in the same application context where it is defined

@Autowired contd…

- @Autowired annotation can be applied on setter methods, constructors and fields.
- Autowired indicating “required dependencies”.
- Autowire **will fail** if no matching bean is available in the context.
- @Autowired(required=false) – indicating not a mandatory dependency. Defaults to true. Autowire **will not fail** if no matching bean is available in the context.

```
@Autowired(required=false)
private AccountRepository accountRepository;
```

@Component

- Indicates that the annotated class is a "component"
- Both identify POJOs as Spring Beans
- Removes the need to specify *almost anything* in XML
- Optionally pass it a String, which will be the bean name
- Default bean name is de-capitalized non-qualified

```
@Component
public class TransferServiceImpl implements TransferService
   public TransferServiceImpl(AccountRepository ar) {
      this.accountRepository = ar;
   }
   …
}
```

@Component contd…

- @Component takes a String parameter that names the bean
- Arguably not a best practice to put bean names in your Java code
- **<context:component-scan** base-package="com.accenture.xx.xx.x" **/>** - required in configuration xml to enable annotation scan in mentioned package

```
@Component("myTransferService")
public class TransferServiceImpl implements TransferService
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    } …
  }
```

@Qualifier

- To used on a field or parameter as a qualifier for a beans when autowiring
- Can be used in other annotations to that can be used as qulaifier
- Needed in case multiple instances of the same type exist, one of which needs to be autowired
- Using an @Qualifier annotation you can inject named beans

Specify the bean name of the bean you want to inject

```
@Autowired
@Qualifier("primaryDataSource")
private DataSource dataSource;
```

When to use What

- Start using annotations for small isolated parts of your application (Spring @MVC controllers)
- Annotations are spread across your code base
- XML is centralized in one (or a few) places
- XML for infrastructure and more 'static' beans
- Annotations for frequently changing beans

Spring Core Annotations : See-It

Demonstration :

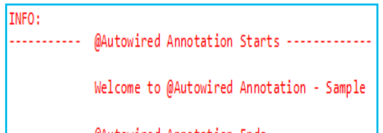
Faculty will demonstrate annotations @Autowired, @Qualifier and @Component

Environment : applicationContext.xml and all files in com.accenture.adfx.module2.sample

Duration: 20 min

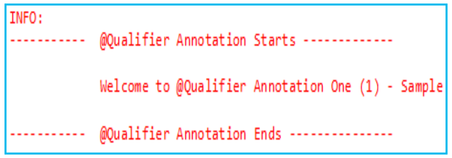
Steps:

1. Open ADFExtensionCodebaseM2SpringCoreAnnotation_participant
2. Open folder com.accenture.adfx.module2.sample
3. Run the following files one by one and check the logs
   - AutowiredSampleClient.java (log 2a)
   - QualifierSampleClient.java (log 2b)
   - ComponentSampleClient.java (log 2c)
4. Refer to their respective main and impl classes too along with applicationContext.xml



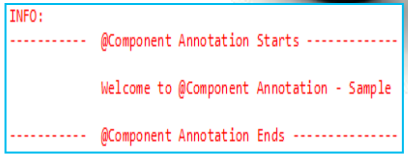
```
INFO:
----------- @Autowired Annotation Starts ------------

            Welcome to @Autowired Annotation - Sample

----------- @Autowired Annotation Ends --------------
```

2a



```
INFO:
----------- @Qualifier Annotation Starts ------------

            Welcome to @Qualifier Annotation One (1) - Sample

----------- @Qualifier Annotation Ends --------------
```

2b



```
INFO:
----------- @Component Annotation Starts ------------

            Welcome to @Component Annotation - Sample

----------- @Component Annotation Ends --------------
```

2c

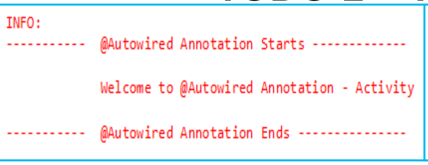
Spring Core Annotations : Try It

Time Allocated: 30 minutes

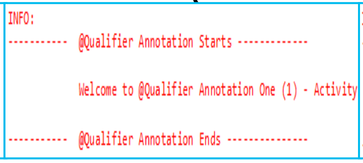
Environment - Eclipse

Steps:

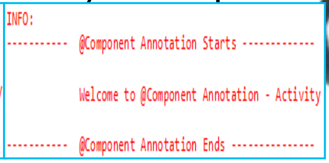
1. Open ADFExtensionCodebaseM2SpringCoreAnnotation_participant
2. Open folder com.accenture.adfx.module2.activity
3. Complete
   - **TODO 1 – TODO4** in applicationContext.xml
   - **TODO 1** in AutowiredActivityMain.java
   - **TODO 1 – TODO3** in AutowiredActivityClient.java
   - **TODO 1** in ComponentActivityMain.java
   - **TODO 1 – TODO3** in ComponentActivityClient.java
   - **TODO 1** in QualifierActivityMain.java
   - **TODO 1 – TODO3** in QualifierActivityClient.java



```
INFO:
-----------  @Autowired Annotation Starts -------------

             Welcome to @Autowired Annotation - Activity

-----------  @Autowired Annotation Ends ---------------
```



```
INFO:
----------  @Qualifier Annotation Starts ------------

            Welcome to @Qualifier Annotation One (1) - Activity

----------  @Qualifier Annotation Ends --------------
```



```
INFO:
----------  @Component Annotation Starts ------------

            Welcome to @Component Annotation - Activity

----------  @Component Annotation Ends --------------
```

Java Based Annotations

- Enables us to write most of the configurations without using XML.
- Uses Annotations instead of XML
- Commonly used annotations are:
  - @Configuration
  - @Bean
  - @Import
  - @Primary
  - @Lazy

@Configuration and @Bean

- Class level annotation that defines a class as a source of bean definitions.
- Uses @Bean annotation to identify a POJO as a Spring Bean

```
@Configuration
public class MyConfiguration{
@Bean
public TestBean testBean(){
    return new TestBean();
}
 }
```

Code is equal to the following XML Declaration

```
<beans>
<bean id =“testBean” class=“com.beans.TestBean/>
</beans>
```

@Import

- @Import annotation is used for importing beans defined in some other Configuration class.

```
@Configuration
public class MyConfiguration1{
@Bean
public TestBean testBean(){
     ….  }}
```

```
@Configuration
@Import(MyConfiguration1.class)
public class MyConfiguration2{
@Bean
public HelloBean helloBean(){
  return new TestBean();
     ….   }}
```

@Primary

- In the same application context, if multiple beans are qualified to autowire a single dependency, we might require to give one bean a preference over other beans.
- @Primary is used for the same!!!
- @Primary has no effect until component-scan is used.

@Primary – Example (Contd..)

```
@Component
public class InvoiceService{
private InvoiceRepository invRepository;
@Autowired
public InvoiceService(InvoiceRepository invRepository)
{
  this.invRepository = invRepository.
}
}

@Component
Public class JDBCRepository
{
  …
}
```

@Primary - Example

```
@Component
@Primary
Public class HibernateRepository { … }
```

- In the above example, since HibernateRepository is annotated with @Primary, Spring will automatically inject this repository over other similar beans equally qualified.

@Lazy

- Indicates whether a bean is to be lazily initalized.
- Used on class directly or indirectly annotated with @Component or on methods annotated with @Bean
- By default bean initialization is eager unless specified explicitly as lazy.

Overview

@Component, so the functionality is the same as we have discussed for @Component, but we annotate classes that are services in the application

@Repository it is a @Component, but we annotate classes that are repositories, so we have the database-related operations in these classes

@Bean it is used to explicitly declare a single bean, rather than letting the framework do it automatically with scanning

Overview

@Autowire this is how we inject a dependency, we do not have to instantiate the class with the ’new’ keyword, it is handler by the framework itself

@Qualitfier there may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use @Qualifier annotation along with @Autowired to remove the confusion by specifying which exact bean will be wired

Overview

@Configuration it is a @Component as well. Indicates that the given class declares one or more @Bean methods. It indicates also that the class may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime

@ComponentScan Spring container can detect and register the beans and the classes with the @Component annotation (as well as the @Services and @Repositorys). We can define the package name in which Spring is going to scan for the components.

Overview

@EnableAutoConfiguration auto-configures the beans that are present in the classpath. For example, if we have tomcat-embedded.jar in the classpath, we need the Tomcat Embedded ServletContainer Factory bean to configure the tomcat server. This will be done because of this annotation

Overview

@SpringBootApplication because most of the applications needs the @Configuration, @ComponentScan and the @EnableAutoConfiguration annotations, they have been merged into this single @EnableAutoConfiguration annotation

Additional Resources

- Additional Links :
  - http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/beans/factory/annotation/

Course / Module Summary

Fill in the blanks to complete Module Summary

- @Autowired annotation can be applied on _______________, _____________ and ________.
- Using an _____________annotation you can inject named beans.
- ___________ annotation takes a String parameter that names the bean

Course / Module Summary

Fill in the blanks to complete Module Summary

- @Autowired annotation can be applied on setter methods, constructors and fields.
- Using an @Qualifier annotation you can inject named beans.
- @Component annotation takes a String parameter that names the bean

Spring Life Cycle

- When a bean is initialized it might require to perform some action before it can come into a usable state(State in which application can use it).
- When a bean is getting destroyed, there might be some cleanup activity required for the given bean. These activities are known as bean Lifecycle.

Spring Life Cycle Activities

- Standard bean lifecycle interfaces & the standard order of execution are given below.
  1- IoC container will look for the *configuration metadata* of given Bean.
  2- Once found, the container will create the instance of Bean(Using reflection API).
  3- After instance creation dependency will be injected(DI).

Spring Life Cycle Activities

- **If Bean Class implements any of the below-highlighted interfaces then the corresponding method will be invoked in below order(Point 4 – 13).**
- **4-** setBeanName method of ***BeanNameAware*** **I**nterface. It sets the name of the bean in the bean factory that created this bean.
- **5-** setBeanClassLoader method of ***BeanClassLoaderAware*** Interface. Callback that supplies the bean to a bean instance.

Spring Life Cycle Activities

- **6-** setBeanFactory method of ***BeanFactoryAware*** Interface. Callback that supplies the owning factory to a bean instance.
- **Below method execution will be applicable when running in an application context. (Points 7 – 11)**
- **7-** setResourceLoader method of ***ResourceLoaderAware*** Interface. It set the ResourceLoader that this object runs in.
- 8- setApplicationEventPublisher method of ***ApplicationEventPublisherAware*** Interface. Set the ApplicationEventPublisher that this object runs in.

Spring Life Cycle Activities

- 9- setMessageSource method of ***MessageSourceAware*** Interface. Set the MessageSource that this object runs in.
- 10- setApplicationContext method of ***ApplicationContextAware*** Interface. Set the ApplicationContext that this object runs in.
- 11- setServletContext method of ***ServletContextAware*** Interface. Set the ServletContext that this object runs in.

Spring Life Cycle Activities

- 12- postProcessBeforeInitialization method of ***BeanPostProcessor*** Interface. Apply this BeanPostProcessor to the given new bean instance before any bean initialization callbacks.
- 13- afterPropertiesSet method of ***InitializingBean*** Interface. Invoked by a BeanFactory after it has set all bean properties supplied.

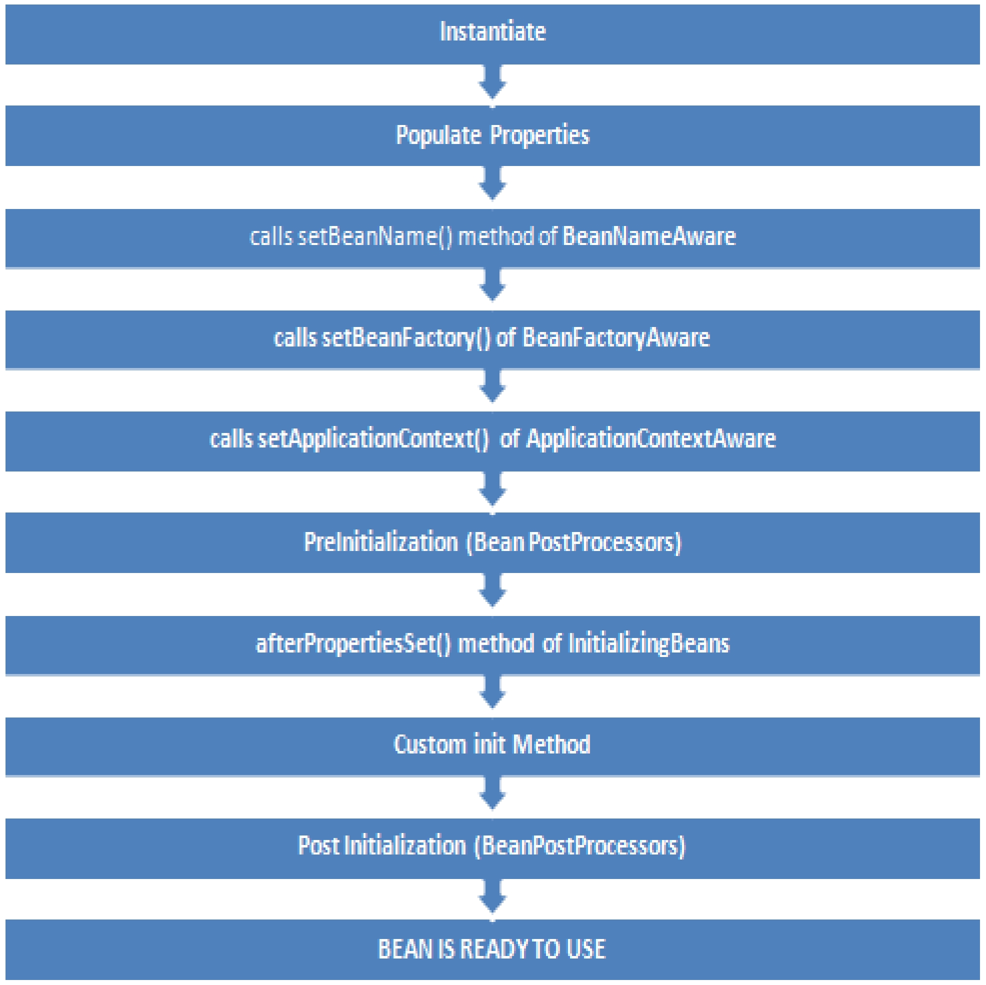
Spring Life Cycle Activities

- **In case Bean class has custom init method defined(via init-method attribute)**
- 14- Custom **init method** will be invoked.
- 15- postProcessAfterInitialization methods of *BeanPostProcessors*. Apply this BeanPostProcessor to the given new bean instance after any bean initialization callbacks

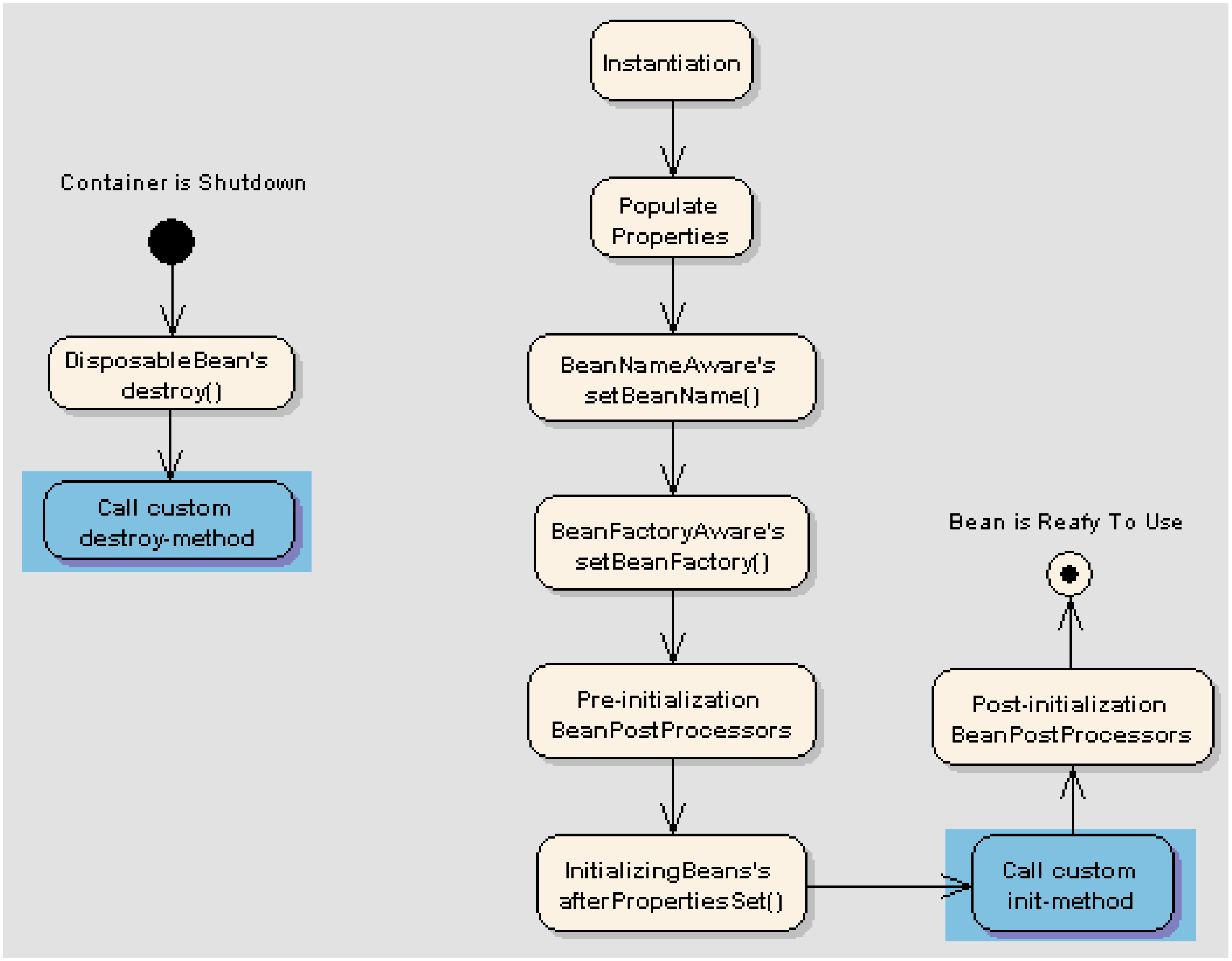
Spring Life Cycle Activities

- When Bean Factory is getting shut down following lifecycle methods will be executed.
- 1- DisposableBean's **destroy** method. Invoked by a BeanFactory on the destruction of a singleton.
  2- **Custome destroy** method will be executed if there is any defined via destroy-method attributes

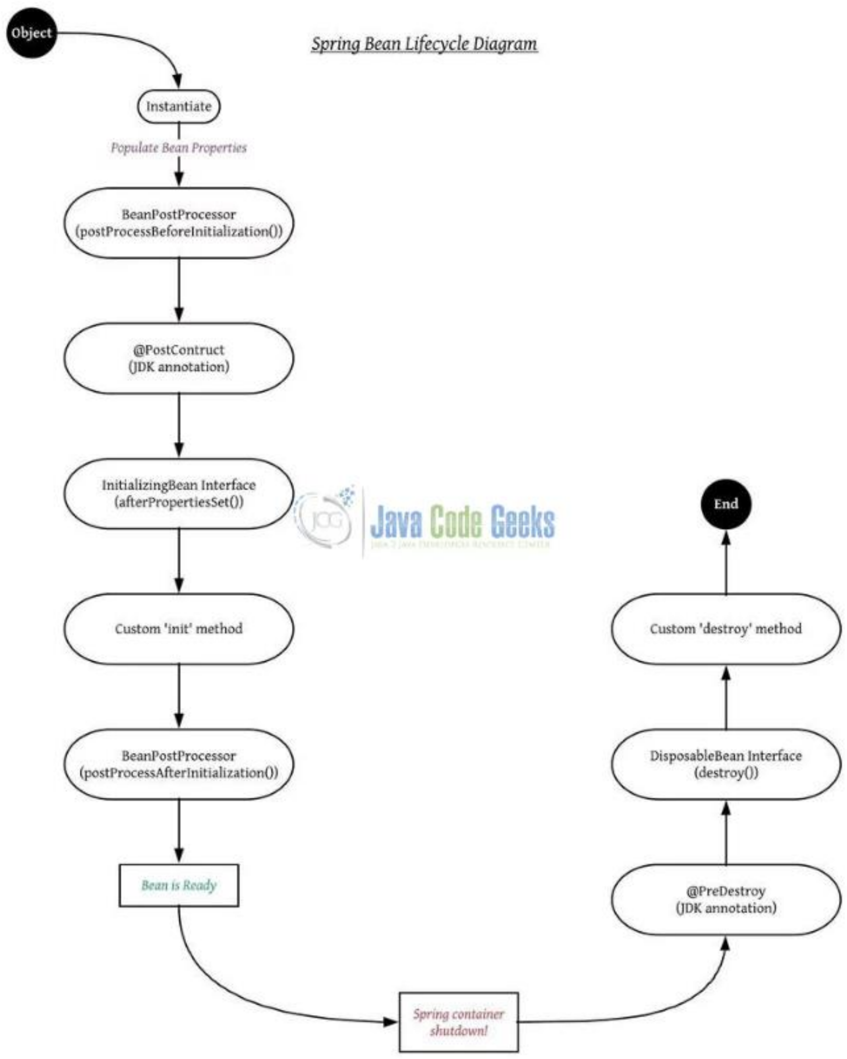
Spring Life Cycle Activities

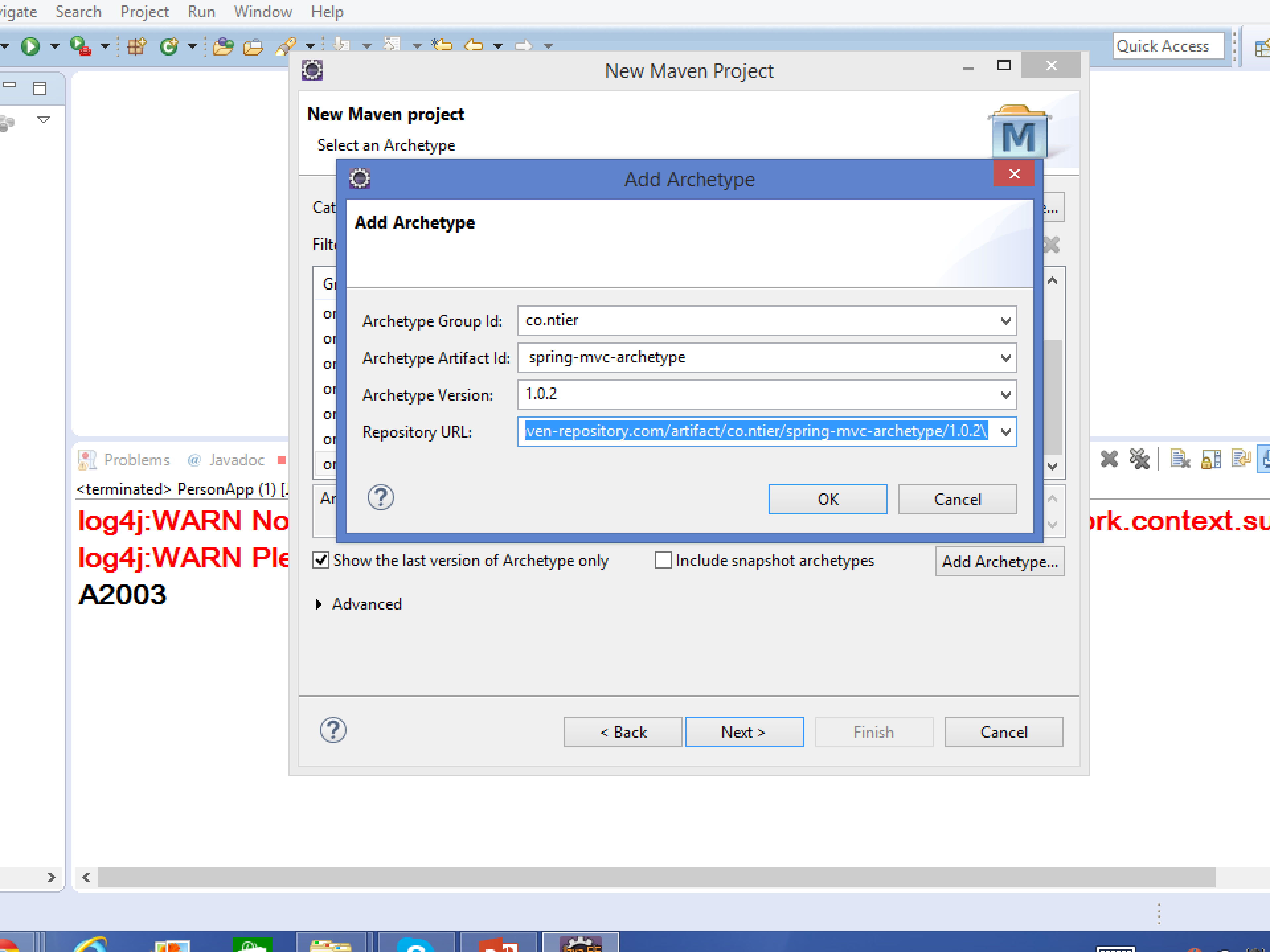

Instantiate
Populate Properties
calls setBeanName() method of BeanNameAware
calls setBeanFactory() of BeanFactoryAware
calls setApplicationContext() of ApplicationContextAware
PreInitialization (Bean PostProcessors)
afterPropertiesSet() method of InitializingBeans
Custom init Method
Post Initialization (BeanPostProcessors)
BEAN IS READY TO USE

Spring Life Cycle Activities


Instantiation
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBeans's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is Reafy To Use
Container is Shutdown
DisposableBean's destroy()
Call custom destroy-method

Spring Life Cycle Activities


Object
Spring Bean Lifecycle Diagram
Instantiate
Populate Bean Properties
BeanPostProcessor (postProcessBeforeInitialization())
@PostContruct (JDK annotation)
InitializingBean Interface (afterPropertiesSet())
Custom 'init' method
BeanPostProcessor (postProcessAfterInitialization())
Bean is Ready
Spring container shutdown!
@PreDestroy (JDK annotation)
DisposableBean Interface (destroy())
Custom 'destroy' method
End
Java Code Geeks



Archetype Group Id: co.ntier
Archetype Artifact Id: spring-mvc-archetype
Archetype Version: 1.0.2
Repository URL: ven-repository.com/artifact/co.ntier/spring-mvc-archetype/1.0.2\

```
<terminated> PersonApp (1) [
log4j:WARN No
log4j:WARN Ple
A2003
```

Course / Module Summary

Now that you have completed this module, you should be familiar with the following concepts:

- Spring's configuration directives can be written in XML or using annotations
- You can mix and match XML and annotations as you please
- @Autowired and @Component allow for almost empty configuration files

Questions