Spring Bean Life Cycle Methods – InitializingBean, DisposableBean, @PostConstruct, @PreDestroy and *Aware interfaces

Spring Beans are the most important part of any Spring application. Spring **ApplicationContext** is responsible to initialize the Spring Beans defined in spring bean configuration file.

Spring Context is also responsible for injection dependencies in the bean, either through setter/constructor methods or by spring autowiring.

Sometimes we want to initialize resources in the bean classes, for example creating database connections or validating third party services at the time of initialization before any client request. Spring framework provide different ways through which we can provide post-initialization and pre-destroy methods in a spring bean.

1. By implementing **InitializingBean** and **DisposableBean** interfaces – Both these interfaces declare a single method where we can initialize/close resources in the bean. For post-initialization, we can implement `InitializingBean` interface and provide implementation of `afterPropertiesSet()` method. For pre-destroy, we can implement `DisposableBean` interface and provide implementation of `destroy()` method. These methods are the callback methods and similar to servlet listener implementations.
   This approach is simple to use but it's not recommended because it will create tight coupling with the Spring framework in our bean implementations.

2. Providing **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

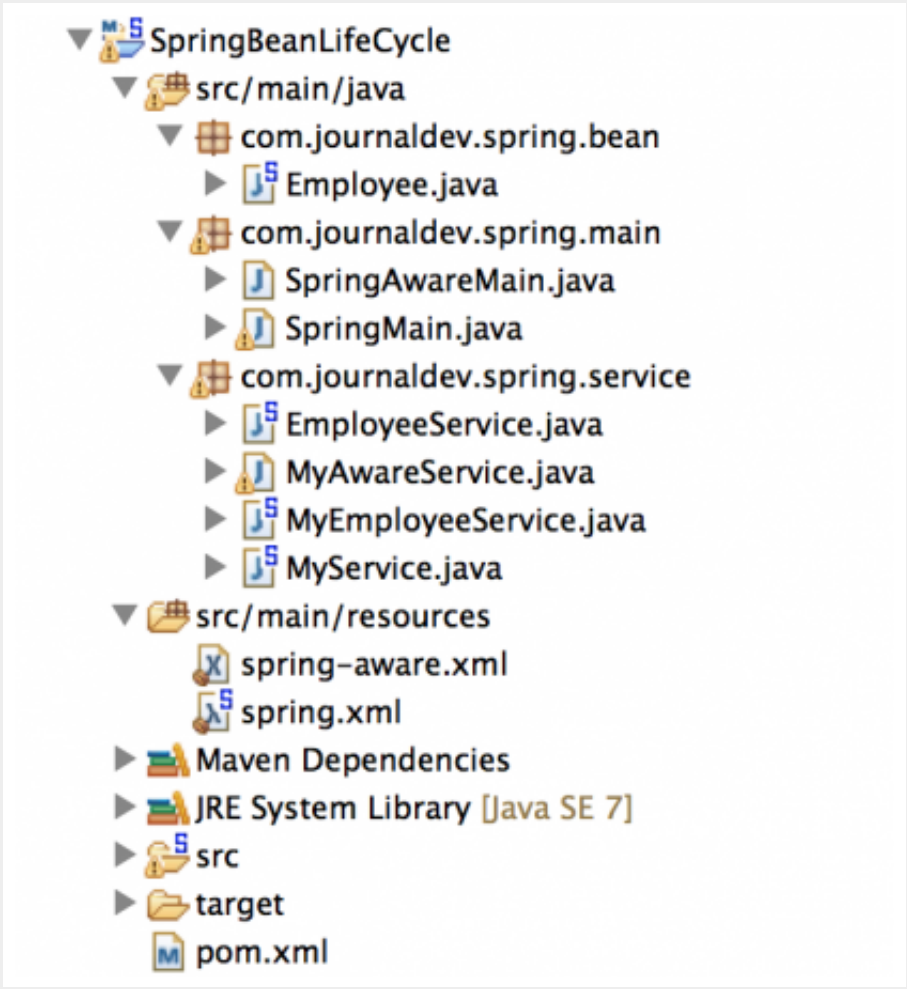
Note that both *post-init* and *pre-destroy* methods should have no arguments but they can throw Exceptions. We would also require to get the bean instance from the spring application context for these methods invocation.

@PostConstruct and @PreDestroy Annotations

Spring framework also support `@PostConstruct` and `@PreDestroy` annotations for defining post-init and pre-destroy methods. These annotations are part of `javax.annotation` package. However for these annotations to work, we need to configure our spring application to look for annotations. We can do this either by defining bean of type

`org.springframework.context.annotation.CommonAnnotationBeanPostProcessor` or by `context:annotation-config` element in spring bean configuration file.

Let's write a simple Spring application to showcase the use of above configurations. Create a Spring Maven project in Spring Tool Suite, final project will look like below image.



Spring Maven Dependencies

We don't need to include any extra dependencies for configuring spring bean life cycle methods, our pom.xml file is like any other standard spring maven project.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.samples</groupId>
  <artifactId>SpringBeanLifeCycle</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>

        <!-- Generic properties -->
        <java.version>1.7</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEnco

        <!-- Spring -->
        <spring-framework.version>4.0.2.RELEASE</spring-framework.version>
```

```
        <!-- Logging -->
        <logback.version>1.0.13</logback.version>
        <slf4j.version>1.7.5</slf4j.version>

    </properties>

    <dependencies>
        <!-- Spring and Transactions -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring-framework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring-framework.version}</version>
        </dependency>

        <!-- Logging with SLF4J & LogBack -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>${logback.version}</version>
            <scope>runtime</scope>
        </dependency>

    </dependencies>
</project>
```

Model Class

Let's create a simple java bean class that will be used in service classes.

Employee.java

```
package com.journaldev.spring.bean;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

InitializingBean and DisposableBean Example

Let's create a service class where we will implement both the interfaces for post-init and pre-destroy methods.

EmployeeService.java

```java
package com.journaldev.spring.service;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

import com.journaldev.spring.bean.Employee;

public class EmployeeService implements InitializingBean, DisposableBean{

    private Employee employee;

    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }

    public EmployeeService(){
        System.out.println("EmployeeService no-args constructor called");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("EmployeeService Closing resources");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("EmployeeService initializing to dummy value");
        if(employee.getName() == null){
            employee.setName("Pankaj");
        }
    }
}
```

Service class with custom post-init and pre-destroy methods

Since we don't want our services to have direct spring framework dependency, let's create another form of Employee Service class where we will have post-init and pre-destroy methods and we will configure them in the spring bean configuration file.

MyEmployeeService.java

```java
package com.journaldev.spring.service;

import com.journaldev.spring.bean.Employee;

public class MyEmployeeService{

    private Employee employee;

    public Employee getEmployee() {
        return employee;
```

```
    }

    public void setEmployee(Employee employee) {
        this.employee = employee;
    }

    public MyEmployeeService(){
        System.out.println("MyEmployeeService no-args constructor called");
    }

    //pre-destroy method
    public void destroy() throws Exception {
        System.out.println("MyEmployeeService Closing resources");
    }

    //post-init method
    public void init() throws Exception {
        System.out.println("MyEmployeeService initializing to dummy value");
        if(employee.getName() == null){
            employee.setName("Pankaj");
        }
    }
}
```

We will look into the spring bean configuration file in a bit. Before that let's create another service class that will use @PostConstruct and @PreDestroy annotations.

@PostConstruct and @PreDestroy Example

Below is a simple class that will be configured as spring bean and for post-init and pre-destroy methods, we are using @PostConstruct and @PreDestroy annotations.

MyService.java

```
package com.journaldev.spring.service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyService {

    @PostConstruct
    public void init(){
        System.out.println("MyService init method called");
    }

    public MyService(){
        System.out.println("MyService no-args constructor called");
    }

    @PreDestroy
    public void destory(){
        System.out.println("MyService destroy method called");
    }
}
```

Spring Bean Configuration File

Let's see how we will configure our beans in spring context file.

spring.xml

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

<!-- Not initializing employee name variable-->
<bean name="employee" class="com.journaldev.spring.bean.Employee" />

<bean name="employeeService" class="com.journaldev.spring.service.EmployeeSer
    <property name="employee" ref="employee"></property>
</bean>

<bean name="myEmployeeService" class="com.journaldev.spring.service.MyEmploye
        init-method="init" destroy-method="destroy">
    <property name="employee" ref="employee"></property>
</bean>

<!-- initializing CommonAnnotationBeanPostProcessor is same as context:annotat
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostPr
<bean name="myService" class="com.journaldev.spring.service.MyService" />
</beans>
```

Notice that I am not initializing employee name in it's bean definition. Since EmployeeService is using interfaces, we don't need any special configuration here.

For MyEmployeeService bean, we are using init-method and destroy-method attributes to let spring framework know our custom methods to execute.

MyService bean configuration doesn't have anything special, but as you can see that I am enabling annotation based configuration for this.

Our application is ready, let's write a test program to see how different methods get executed.

Test Program

SpringMain.java

```
package com.journaldev.spring.main;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.service.EmployeeService;
import com.journaldev.spring.service.MyEmployeeService;

public class SpringMain {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationConte

        System.out.println("Spring Context initialized");

        //MyEmployeeService service = ctx.getBean("myEmployeeService", MyEmplo
```

```
        EmployeeService service = ctx.getBean("employeeService", EmployeeServi
        System.out.println("Bean retrieved from Spring Context");
        System.out.println("Employee Name="+service.getEmployee().getName());
        ctx.close();
        System.out.println("Spring Context Closed");
    }
}
```

When we run above test program, we get below output.

```
Apr 01, 2014 10:50:50 PM org.springframework.context.support.ClassPathXmlAppli
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationC
Apr 01, 2014 10:50:50 PM org.springframework.beans.factory.xml.XmlBeanDefinit
INFO: Loading XML bean definitions from class path resource [spring.xml]
EmployeeService no-args constructor called
EmployeeService initializing to dummy value
MyEmployeeService no-args constructor called
MyEmployeeService initializing to dummy value
MyService no-args constructor called
MyService init method called
Spring Context initialized
Bean retrieved from Spring Context
Employee Name=Pankaj
Apr 01, 2014 10:50:50 PM org.springframework.context.support.ClassPathXmlAppli
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationConte
MyService destroy method called
MyEmployeeService Closing resources
EmployeeService Closing resources
Spring Context Closed
```

Important Points:

- From the console output it's clear that Spring Context is first using no-args constructor to initialize the bean object and then calling the post-init method.
- The order of bean initialization is same as it's defined in the spring bean configuration file.
- The context is returned only when all the spring beans are initialized properly with post-init method executions.
- Employee name is printed as "Pankaj" because it was initialized in the post-init method.
- When context is getting closed, beans are destroyed in the reverse order in which they were initialized i.e in LIFO (Last-In-First-Out) order.

You can uncomment the code to get bean of type `MyEmployeeService` and confirm that output will be similar and follow all the points mentioned above.

Spring Aware Interfaces

Sometimes we need Spring Framework objects in our beans to perform some operations, for

example reading ServletConfig and ServletContext parameters or to know the bean definitions loaded by the ApplicationContext. That's why spring framework provides a bunch of *Aware interfaces that we can implement in our bean classes.

`org.springframework.beans.factory.Aware` is the root marker interface for all these Aware interfaces. All of the *Aware interfaces are sub-interfaces of Aware and declare a single setter method to be implemented by the bean. Then spring context uses setter-based dependency injection to inject the corresponding objects in the bean and make it available for our use.

Spring Aware interfaces are similar to servlet listeners with callback methods and implementing observer design pattern.

Some of the important Aware interfaces are:

- **ApplicationContextAware** – to inject ApplicationContext object, example usage is to get the array of bean definition names.
- **BeanFactoryAware** – to inject BeanFactory object, example usage is to check scope of a bean.
- **BeanNameAware** – to know the bean name defined in the configuration file.
- **ResourceLoaderAware** – to inject ResourceLoader object, example usage is to get the input stream for a file in the classpath.
- **ServletContextAware** – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.
- **ServletConfigAware** – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

Let's see these Aware interfaces usage in action by implementing few of them in a class that we will configure as spring bean.

MyAwareService.java

```
1  package com.journaldev.spring.service;
2
3  import java.util.Arrays;
4
5  import org.springframework.beans.BeansException;
6  import org.springframework.beans.factory.BeanClassLoaderAware;
7  import org.springframework.beans.factory.BeanFactory;
8  import org.springframework.beans.factory.BeanFactoryAware;
9  import org.springframework.beans.factory.BeanNameAware;
10 import org.springframework.context.ApplicationContext;
11 import org.springframework.context.ApplicationContextAware;
12 import org.springframework.context.ApplicationEventPublisher;
13 import org.springframework.context.ApplicationEventPublisherAware;
14 import org.springframework.context.EnvironmentAware;
15 import org.springframework.context.ResourceLoaderAware;
16 import org.springframework.context.annotation.ImportAware;
17 import org.springframework.core.env.Environment;
18 import org.springframework.core.io.Resource;
19 import org.springframework.core.io.ResourceLoader;
20 import org.springframework.core.type.AnnotationMetadata;
```

```
21
22  public class MyAwareService implements ApplicationContextAware,
23          ApplicationEventPublisherAware, BeanClassLoaderAware, BeanFactoryAware
24          BeanNameAware, EnvironmentAware, ImportAware, ResourceLoaderAware {
25
26      @Override
27      public void setApplicationContext(ApplicationContext ctx)
28              throws BeansException {
29          System.out.println("setApplicationContext called");
30          System.out.println("setApplicationContext:: Bean Definition Names="
31                  + Arrays.toString(ctx.getBeanDefinitionNames()));
32      }
33
34      @Override
35      public void setBeanName(String beanName) {
36          System.out.println("setBeanName called");
37          System.out.println("setBeanName:: Bean Name defined in context="
38                  + beanName);
39      }
40
41      @Override
42      public void setBeanClassLoader(ClassLoader classLoader) {
43          System.out.println("setBeanClassLoader called");
44          System.out.println("setBeanClassLoader:: ClassLoader Name="
45                  + classLoader.getClass().getName());
46      }
47
48      @Override
49      public void setResourceLoader(ResourceLoader resourceLoader) {
50          System.out.println("setResourceLoader called");
51          Resource resource = resourceLoader.getResource("classpath:spring.xml")
52          System.out.println("setResourceLoader:: Resource File Name="
53                  + resource.getFilename());
54      }
55
56      @Override
57      public void setImportMetadata(AnnotationMetadata annotationMetadata) {
58          System.out.println("setImportMetadata called");
59      }
60
61      @Override
62      public void setEnvironment(Environment env) {
63          System.out.println("setEnvironment called");
64      }
65
66      @Override
67      public void setBeanFactory(BeanFactory beanFactory) throws BeansException
68          System.out.println("setBeanFactory called");
69          System.out.println("setBeanFactory:: employee bean singleton="
70                  + beanFactory.isSingleton("employee"));
71      }
72
73      @Override
74      public void setApplicationEventPublisher(
75              ApplicationEventPublisher applicationEventPublisher) {
76          System.out.println("setApplicationEventPublisher called");
77      }
78
79  }
```

Spring Bean Configuration File

Very simple spring bean configuration file.

spring-aware.xml

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

<bean name="employee" class="com.journaldev.spring.bean.Employee" />

<bean name="myAwareService" class="com.journaldev.spring.service.MyAwareServi

</beans>
```

Spring *Aware Test Program

SpringAwareMain.java

```
package com.journaldev.spring.main;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.service.MyAwareService;

public class SpringAwareMain {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationConte

        ctx.getBean("myAwareService", MyAwareService.class);

        ctx.close();
    }

}
```

Now when we execute above class, we get following output.

```
Apr 01, 2014 11:27:05 PM org.springframework.context.support.ClassPathXmlAppli
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationC
Apr 01, 2014 11:27:05 PM org.springframework.beans.factory.xml.XmlBeanDefinit
INFO: Loading XML bean definitions from class path resource [spring-aware.xml]
setBeanName called
setBeanName:: Bean Name defined in context=myAwareService
setBeanClassLoader called
setBeanClassLoader:: ClassLoader Name=sun.misc.Launcher$AppClassLoader
setBeanFactory called
setBeanFactory:: employee bean singleton=true
setEnvironment called
setResourceLoader called
setResourceLoader:: Resource File Name=spring.xml
setApplicationEventPublisher called
setApplicationContext called
setApplicationContext:: Bean Definition Names=[employee, myAwareService]
Apr 01, 2014 11:27:05 PM org.springframework.context.support.ClassPathXmlAppli
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationConte
```

Console output of the test program is simple to understand, I won't go into much detail about that.

That's all for the Spring Bean life cycle methods and injecting framework specific objects into the spring beans. Please download sample project from below link and analyze it to learn more about them.