JournalDev

Spring Framework Tutorial

Pankaj Kumar

Table of Content

1. Dependency Injection Design Pattern in Java

Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement dependency injection pattern to move the dependency resolution from compile-time to runtime.

Dependency injection pattern seems hard to grasp with theory, so I would take a simple example and then we will see how to use dependency injection pattern to achieve loose coupling and extendibility in the application.

Let's say we have an application where we consume *EmailService* to send emails. Normally we would implement this like below.

```
package com.journaldev.singleton;

public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton instance = new
EagerInitializedSingleton();

    //private constructor to avoid client applications to use
constructor
    private EagerInitializedSingleton(){}

    public static EagerInitializedSingleton getInstance(){
        return instance;
    }
}
```

EmailService class holds the logic to send email message to the recipient email address. Our application code will be like below.

```
package com.journaldev.java.legacy;

public class MyApplication {

    private EmailService email = new EmailService();

    public void processMessages(String msg, String rec){
        //do some msg validation, manipulation logic etc
        this.email.sendEmail(msg, rec);
    }
}
```

Our client code that will use *MyApplication* class to send email messages will be like below.

```
package com.journaldev.java.legacy;

public class MyLegacyTest {

    public static void main(String[] args) {
        MyApplication app = new MyApplication();
        app.processMessages("Hi Pankaj", "pankaj@abc.com");
    }

}
```

At first look, there seems nothing wrong with above implementation. But above code logic has certain limitations.

- MyApplication class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even harder.
- If we want to extend our application to provide additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
- Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

One can argue that we can remove the email service instance creation from MyApplication class by having a constructor that requires email service as argument.

```java
package com.journaldev.java.legacy;

public class MyApplication {

    private EmailService email = null;

    public MyApplication(EmailService svc){
        this.email=svc;
    }

    public void processMessages(String msg, String rec){
        //do some msg validation, manipulation logic etc
        this.email.sendEmail(msg, rec);
    }
}
```

But in this case, we are asking client applications or test classes to initializing the email service that is not a good design decision.

Now let’s see how we can apply dependency injection pattern to solve all the problems with above implementation. Dependency Injection pattern requires at least following:

1. Service components should be designed with base class or interface. It’s better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

A. Service Components

For our case, we can have *MessageService* that will declare the contract for service implementations.

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

public interface MessageService {

    void sendMessage(String msg, String rec);
}
```

Now let’s say we have Email and SMS services that implement above interfaces.

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

public class EmailServiceImpl implements MessageService {

    @Override
    public void sendMessage(String msg, String rec) {
        //logic to send email
        System.out.println("Email sent to "+rec+ " with Message="+msg);
    }

}
```

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

public class SMSServiceImpl implements MessageService {

    @Override
    public void sendMessage(String msg, String rec) {
        //logic to send SMS
        System.out.println("SMS sent to "+rec+ " with Message="+msg);
    }

}
```

Our services are ready and now we can write our consumer class.

B. Service Consumer

We are not required to have base interfaces for consumer classes but I will have a Consumer interface declaring contract for consumer classes.

```
package com.journaldev.java.dependencyinjection.consumer;

public interface Consumer {

    void processMessages(String msg, String rec);
}
```

My consumer class implementation is like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

    private MessageService service;

    public MyDIApplication(MessageService svc){
        this.service=svc;
    }

    @Override
    public void processMessages(String msg, String rec){
        //do some msg validation, manipulation logic etc
        this.service.sendMessage(msg, rec);
    }

}
```

Notice that our application class is just using the service. It does not initialize the service that leads to better "*separation of concerns*". Also use of service interface allows us to easily test the application by mocking the MessageService and bind the services at runtime rather than compile time.
Now we are ready to write injector classes that will initialize the service and also consumer classes.

C. Injectors Classes

Let’s have an interface *MessageServiceInjector* with method declaration that returns the Consumer class.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

public interface MessageServiceInjector {

    public Consumer getConsumer();
}
```

Now for every service, we will have to create injector classes like below.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import
com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import
com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

    @Override
    public Consumer getConsumer() {
        return new MyDIApplication(new EmailServiceImpl());
    }

}
```

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import
com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.SMSServiceImpl;

public class SMSServiceInjector implements MessageServiceInjector {

    @Override
    public Consumer getConsumer() {
        return new MyDIApplication(new SMSServiceImpl());
    }
}
```

Now let’s see how our client applications will use the application with a simple program.

```
package com.journaldev.java.dependencyinjection.test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import
com.journaldev.java.dependencyinjection.injector.EmailServiceInjector;
import
com.journaldev.java.dependencyinjection.injector.MessageServiceInjector
;
import
com.journaldev.java.dependencyinjection.injector.SMSServiceInjector;

public class MyMessageDITest {

    public static void main(String[] args) {
        String msg = "Hi Pankaj";
        String email = "pankaj@abc.com";
        String phone = "4088888888";
        MessageServiceInjector injector = null;
        Consumer app = null;

        //Send email
        injector = new EmailServiceInjector();
        app = injector.getConsumer();
        app.processMessages(msg, email);

        //Send SMS
        injector = new SMSServiceInjector();
        app = injector.getConsumer();
        app.processMessages(msg, phone);
    }

}
```

As you can see that our application classes are responsible only for using the service. Service classes are created in injectors. Also if we have to further extend our application to allow Facebook messaging, we will have to write Service classes and injector classes only.

So dependency injection implementation solved the problem with hard-coded dependency and helped us in making our application flexible and easy to extend. Now let’s see how easily we can test our application class by mocking the injector and service classes.

D. JUnit Test Case with Mock Injector and Service

```
package com.journaldev.java.dependencyinjection.test;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import
com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import
com.journaldev.java.dependencyinjection.injector.MessageServiceInjector
;
import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplicationJUnitTest {

    private MessageServiceInjector injector;
    @Before
    public void setUp(){
        //mock the injector with anonymous class
        injector = new MessageServiceInjector() {

            @Override
            public Consumer getConsumer() {
                //mock the message service
                return new MyDIApplication(new MessageService() {

                    @Override
                    public void sendMessage(String msg, String rec) {
                        System.out.println("Mock Message Service
implementation");

                    }
                });
            }
        };
    }

    @Test
    public void test() {
        Consumer consumer = injector.getConsumer();
        consumer.processMessages("Hi Pankaj", "pankaj@abc.com");
    }

    @After
    public void tear(){
        injector = null;
    }

}
```

As you can see that I am using anonymous classes to *mock the injector and service classes* and I can easily test my application methods. I am using JUnit 4 for above test class, so make sure it's in your project build path if you are running above test class.

We have used constructors to inject the dependencies in the application classes, another way is to use setter method to inject dependencies in application classes. For setter method dependency injection, our application class will be implemented like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

    private MessageService service;

    public MyDIApplication(){}

    //setter dependency injection
    public void setService(MessageService service) {
        this.service = service;
    }
    @Override
    public void processMessages(String msg, String rec){
        //do some msg validation, manipulation logic etc
        this.service.sendMessage(msg, rec);
    }
}


package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import
com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import
com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

    @Override
    public Consumer getConsumer() {
        MyDIApplication app = new MyDIApplication();
        app.setService(new EmailServiceImpl());
        return app;
    }
}
```

One of the best example of setter dependency injection is Struts2 Servlet API Aware interfaces.

Whether to use Constructor based dependency injection or setter based is a design decision and depends on your requirements. For example, if my application can’t work at all without the service class then I would prefer constructor based DI or else I would go for setter method based DI to use it only when it’s really needed.

Dependency Injection is a way to achieve **Inversion of control** (**IoC**) in our application by moving objects binding from compile time to runtime. We can achieve IoC through Factory Pattern, Template Method Design Pattern, Strategy Pattern and Service Locator pattern too.

Spring, **Google Guice** and **Java EE CDI** frameworks facilitate the process of dependency injection through use of Java Reflection API and java annotations. All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

E. Benefits of Dependency Injection

Some of the benefits of using Dependency Injection are:

- Separation of Concerns
- Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component
- Configurable components makes application easily extendable
- Unit testing is easy with mock objects

F. Disadvantages of Dependency Injection

Dependency injection has some disadvantages too:

- If overused, it can lead to maintenance issues because effect of changes are known at runtime.
- Dependency injection hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

You can download the source code of "Dependency Injection Project" from here.

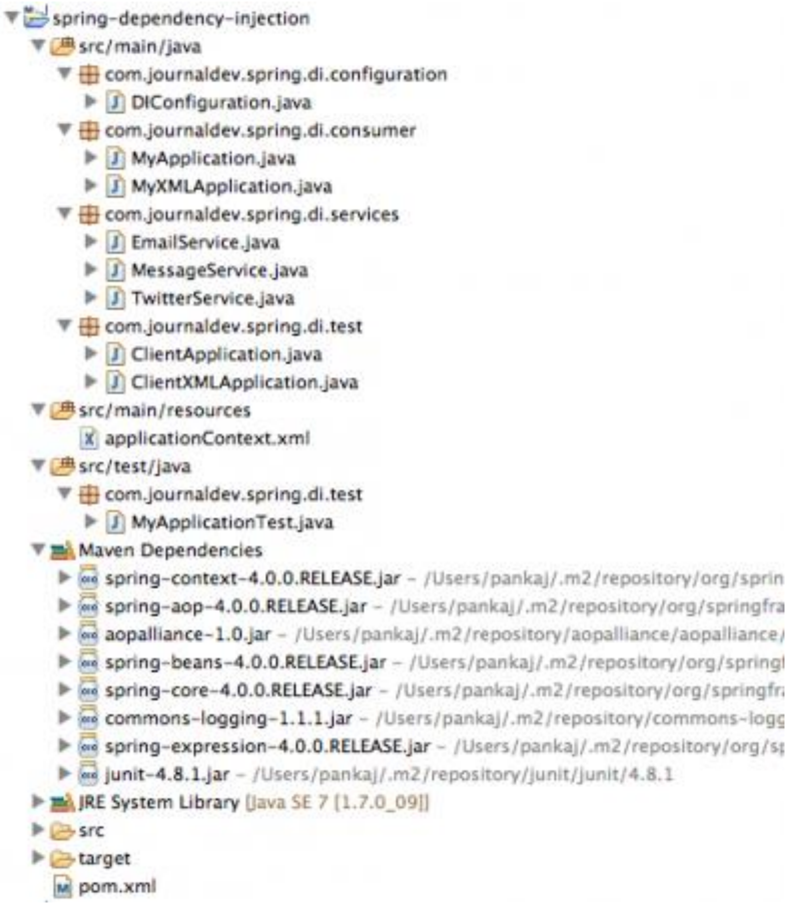
That's all for dependency injection pattern in java. It's good to know and use it when we are in control of the services.

2. Spring Dependency Injection Example

Spring Framework core concepts are “**Dependency Injection**” and “**Aspect Oriented Programming**“. We learned the benefits of Dependency Injection in above section and how we can implement it in Java.

This tutorial section is aimed to provide dependency injection example in Spring Framework with both annotation based configuration and XML file based configuration. I will also provide JUnit test case example for the application, since easy testability is one of the major benefits of dependency injection.

I have created *spring-dependency-injection* maven project whose structure looks like below image.



```
▼ spring-dependency-injection
  ▼ src/main/java
    ▼ com.journaldev.spring.di.configuration
      ▶ DIConfiguration.java
    ▼ com.journaldev.spring.di.consumer
      ▶ MyApplication.java
      ▶ MyXMLApplication.java
    ▼ com.journaldev.spring.di.services
      ▶ EmailService.java
      ▶ MessageService.java
      ▶ TwitterService.java
    ▼ com.journaldev.spring.di.test
      ▶ ClientApplication.java
      ▶ ClientXMLApplication.java
  ▼ src/main/resources
      applicationContext.xml
  ▼ src/test/java
    ▼ com.journaldev.spring.di.test
      ▶ MyApplicationTest.java
  ▼ Maven Dependencies
    ▶ spring-context-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/sprin
    ▶ spring-aop-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/springfra
    ▶ aopalliance-1.0.jar - /Users/pankaj/.m2/repository/aopalliance/aopalliance/
    ▶ spring-beans-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/spring
    ▶ spring-core-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/springfr
    ▶ commons-logging-1.1.1.jar - /Users/pankaj/.m2/repository/commons-logg
    ▶ spring-expression-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/sp
    ▶ junit-4.8.1.jar - /Users/pankaj/.m2/repository/junit/junit/4.8.1
  ▶ JRE System Library [Java SE 7 [1.7.0_09]]
  ▶ src
  ▶ target
    pom.xml
```

Let’s look at each of the components one by one.

A. Spring Maven Dependencies

I have added Spring and JUnit maven dependencies in pom.xml file, final pom.xml code is below.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.journaldev.spring</groupId>
    <artifactId>spring-dependency-injection</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

Current stable version of Spring Framework is *4.0.0.RELEASE* and JUnit current version is *4.8.1*, if you are using any other versions then there might be a small chance that the project will need some change. If you will build the project, you will notice some other jars are also added to maven dependencies because of transitive dependencies, just like above image.

B. Service Classes

Let’s say we want to send email message and twitter message to the users. For dependency injection, we need to have a base class for the services. So I have MessageService interface with single method declaration for sending message.

```
package com.journaldev.spring.di.services;

public interface MessageService {

    boolean sendMessage(String msg, String rec);
}
```

Now we will have actual implementation classes to send email and twitter message.

```
package com.journaldev.spring.di.services;

public class EmailService implements MessageService {

    public boolean sendMessage(String msg, String rec) {
        System.out.println("Email Sent to "+rec+ " with Message="+msg);
        return true;
    }

}
```

```
package com.journaldev.spring.di.services;

public class TwitterService implements MessageService {

    public boolean sendMessage(String msg, String rec) {
        System.out.println("Twitter message Sent to "+rec+ " with 
Message="+msg);
        return true;
    }

}
```

Now that our services are ready, we can move on to Component classes that will consume the service.

C. Component Classes

Let’s write a consumer class for above services. We will have two consumer classes – one with Spring annotations for **autowiring** and another without annotation and wiring configuration will be provided in the XML configuration file.

```
package com.journaldev.spring.di.consumer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.stereotype.Component;

import com.journaldev.spring.di.services.MessageService;

@Component
public class MyApplication {

    //field-based dependency injection
    //@Autowired
    private MessageService service;

//  constructor-based dependency injection
//  @Autowired
//  public MyApplication(MessageService svc){
//      this.service=svc;
//  }

    @Autowired
    public void setService(MessageService svc){
        this.service=svc;
    }

    public boolean processMessage(String msg, String rec){
        //some magic like validation, logging etc
        return this.service.sendMessage(msg, rec);
    }
}
```

Few important points about MyApplication class:

- *@Compon*ent annotation is added to the class, so that when Spring framework will scan for the components, this class will be treated as component. @Component annotation can be applied only to the class and its retention policy is Runtime. If you are not familiar with Annotations retention policy, I would suggest you to read java annotations tutorial.
- *@Autowired* annotation is used to let Spring know that **autowiring** is required. This can be applied to field, constructor and methods. This annotation allows us to implement constructor-based, field-based or method-based dependency injection in our components.
- For our example, I am using method-based dependency injection. You can uncomment the constructor method to switch to constructor based dependency injection.

Now let’s write similar class without annotations.

```
package com.journaldev.spring.di.consumer;

import com.journaldev.spring.di.services.MessageService;

public class MyXMLApplication {

    private MessageService service;

    //constructor-based dependency injection
//  public MyXMLApplication(MessageService svc) {
//      this.service = svc;
//  }

    //setter-based dependency injection
    public void setService(MessageService svc){
        this.service=svc;
    }

    public boolean processMessage(String msg, String rec) {
        // some magic like validation, logging etc
        return this.service.sendMessage(msg, rec);
    }
}
```

A simple application class consuming the service. For XML based configuration, we can use implement either constructor-based dependency injection or method-based dependency injection. Note that method-based and setter-based injection approaches are same, it’s just that some prefer calling it setter-based and some call it method-based.

D. Spring Configuration with Annotations

For annotation based configuration, we need to write a Configurator class that will be used to inject the actual implementation bean to the component property.

```
package com.journaldev.spring.di.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.journaldev.spring.di.services.EmailService;
import com.journaldev.spring.di.services.MessageService;
```

```
@Configuration
@ComponentScan(value={"com.journaldev.spring.di.consumer"})
public class DIConfiguration {

    @Bean
    public MessageService getMessageService(){
        return new EmailService();
    }
}
```

Some important points related to above class are:

- *@Configuration* annotation is used to let Spring know that it's a Configuration class.
- *@ComponentScan* annotation is used with @Configuration annotation to specify the packages to look for Component classes.
- *@Bean* annotation is used to let Spring framework know that this method should be used to get the bean implementation to inject in Component classes.

Let's write a simple program to test our annotation based Spring Dependency Injection example.

```
package com.journaldev.spring.di.test;

import
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt;

import com.journaldev.spring.di.configuration.DIConfiguration;
import com.journaldev.spring.di.consumer.MyApplication;

public class ClientApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(DIConfiguration.class);
        MyApplication app = context.getBean(MyApplication.class);

        app.processMessage("Hi Pankaj", "pankaj@abc.com");

        //close the context
        context.close();
    }

}
```

AnnotationConfigApplicationContext is the implementation of AbstractApplicationContext abstract class and it's used for autowiring the services to components when annotations are used. AnnotationConfigApplicationContext constructor takes Class as argument that will be used to get the bean implementation to inject in component classes.

getBean(Class) method returns the Component object and uses the configuration for autowiring the objects. Context objects are resource intensive, so we should close them when we are done with it. When we run above program, we get below output.

```
Dec 16, 2013 11:49:20 PM
org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt@3067ed13: startup date [Mon Dec 16 23:49:20 PST 2013]; root of
context hierarchy
Email Sent to pankaj@abc.com with Message=Hi Pankaj
Dec 16, 2013 11:49:20 PM
org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt@3067ed13: start
```

E. Spring XML Based Configuration

We will create Spring configuration file with below data, file name can be anything.

```
<?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://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

<!--
    <bean id="MyXMLApp"
class="com.journaldev.spring.di.consumer.MyXMLApplication">
        <constructor-arg>
```

```
            <bean
class="com.journaldev.spring.di.services.TwitterService" />
        </constructor-arg>
    </bean>
-->
    <bean id="twitter"
class="com.journaldev.spring.di.services.TwitterService"></bean>
    <bean id="MyXMLApp"
class="com.journaldev.spring.di.consumer.MyXMLApplication">
        <property name="service" ref="twitter"></property>
    </bean>
</beans>
```

Notice that above XML contains configuration for both constructor-based and setter-based dependency injection. Since *MyXMLApplication* is using setter method for injection, the bean configuration contains *property* element for injection. For constructor based injection, we have to use *constructor-arg* element.

The configuration XML file is placed in the source directory, so it will be in the classes directory after build.

Let’s see how to use XML based configuration with a simple program.

```
package com.journaldev.spring.di.test;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.di.consumer.MyXMLApplication;

public class ClientXMLApplication {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
                "applicationContext.xml");
        MyXMLApplication app = context.getBean(MyXMLApplication.class);

        app.processMessage("Hi Pankaj", "pankaj@abc.com");

        // close the context
        context.close();
    }

}
```

ClassPathXmlApplicationContext is used to get the ApplicationContext object by providing the configuration files location. It has multiple overloaded constructors and we can provide multiple config files also.

Rest of the code is similar to annotation based configuration test program, the only difference is the way we get the ApplicationContext object based on our configuration choice.

When we run above program, we get following output.

```
Dec 17, 2013 12:01:23 AM
org.springframework.context.support.AbstractApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@4eea
abad: startup date [Tue Dec 17 00:01:23 PST 2013]; root of context
hierarchy
Dec 17, 2013 12:01:23 AM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[applicationContext.xml]
Twitter message Sent to pankaj@abc.com with Message=Hi Pankaj
Dec 17, 2013 12:01:23 AM
org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing
org.springframework.context.support.ClassPathXmlApplicationContext@4eea
abad: startup date [Tue Dec 17 00:01:23 PST 2013]; root of context
hierarchy
```

Notice that some of the output is written by Spring Framework. Since Spring Framework uses log4j for logging purpose and I have not configured it, the output is getting written to console.

F. Spring JUnit Test Case

One of the major benefit of dependency injection is the ease of having mock service classes rather than using actual services. So I have combined all of the learning from above and written everything in a single JUnit 4 test class.

```
package com.journaldev.spring.di.test;

import org.junit.Assert;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.journaldev.spring.di.consumer.MyApplication;
import com.journaldev.spring.di.services.MessageService;

@Configuration
@ComponentScan(value="com.journaldev.spring.di.consumer")
public class MyApplicationTest {

    private AnnotationConfigApplicationContext context = null;

    @Bean
    public MessageService getMessageService() {
        return new MessageService(){

            public boolean sendMessage(String msg, String rec) {
                System.out.println("Mock Service");
                return true;
            }

        };
    }

    @Before
    public void setUp() throws Exception {
        context = new
AnnotationConfigApplicationContext(MyApplicationTest.class);
    }

    @After
    public void tearDown() throws Exception {
        context.close();
    }

    @Test
    public void test() {
        MyApplication app = context.getBean(MyApplication.class);
        Assert.assertTrue(app.processMessage("Hi Pankaj",
"pankaj@abc.com"));
    }

}
```

The class is annotated with @Configuration and @ComponentScan annotation because *getMessageService()* method returns the MessageService mock implementation. That's why *getMessageService()* is annotated with @Bean annotation.

Since I am testing MyApplication class that is configured with annotation, I am using AnnotationConfigApplicationContext and creating it's object in the setUp() method. The context is getting closed in *tearDown()* method. *test()* method code is just getting the component object from context and testing it.

3. Spring IoC Container and Spring Bean

Spring Framework is built on the **Inversion of Control** (**IOC**) principle. Dependency injection is the technique to implement IoC in applications. This article is aimed to explain core concepts of Spring IoC container and Spring Bean with example programs.

1. Spring Ioc Container
2. Spring Bean
3. Spring Bean Scopes
4. Spring Bean Configuration
5. Spring IoC and Bean Example Project
    1. XML Based Bean Configuration
    2. Annotation Based Bean Configuration
    3. Java Based Bean Configuration

A. Spring IoC Container

Inversion of Control is the mechanism to achieve loose-coupling between Objects dependencies. To achieve loose coupling and dynamic binding of the objects at runtime, the objects define their dependencies that are being injected by other assembler objects. Spring IoC container is the program that *injects* dependencies into an object and make it ready for our use. We have already looked how we can use Spring Dependency Injection to implement IoC in our applications.

Spring Framework IoC container classes are part of org.springframework.beans and org.springframework.context packages and provides us different ways to decouple the object dependencies.

BeanFactory is the root interface of Spring IoC container. ApplicationContext is the child interface of BeanFactory interface that provide Spring's AOP features, internationalization etc. Some of the useful child-interfaces of ApplicationContext are ConfigurableApplicationContext and WebApplicationContext. Spring Framework provides a number of useful ApplicationContext implementation classes that we can use to get the context and then the Spring Bean.

Some of the useful ApplicationContext implementations that we use are;

- **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
- **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
- **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

Usually if you are working on Spring MVC application and your application is configured to use Spring Framework, Spring IoC container gets initialized when application starts and when a bean is requested, the dependencies are injected automatically.

However for standalone application, you need to initialize the container somewhere in the application and then use it to get the spring beans.

B. Spring Bean

Spring Bean is nothing special, any object in the Spring framework that we initialize through Spring container is called Spring Bean. Any normal Java POJO class can be a Spring Bean if it’s configured to be initialized via container by providing configuration metadata information.

C. Spring Bean Scopes

There are five scopes defined for Spring Beans.

1. **singleton** – Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues.
2. **prototype** – A new instance will be created every time the bean is requested.
3. **request** – This is same as prototype scope, however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **session** – A new bean will be created for each HTTP session by the container.
5. **global-session** – This is used to create global session beans for Portlet applications.

Spring Framework is extendable and we can create our own scopes too, however most of the times we are good with the scopes provided by the framework.

D. Spring Bean Configuration

Spring Framework provide three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** – By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** – By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** – Starting from Spring 3.0, we can configure Spring beans using java programs. Some important

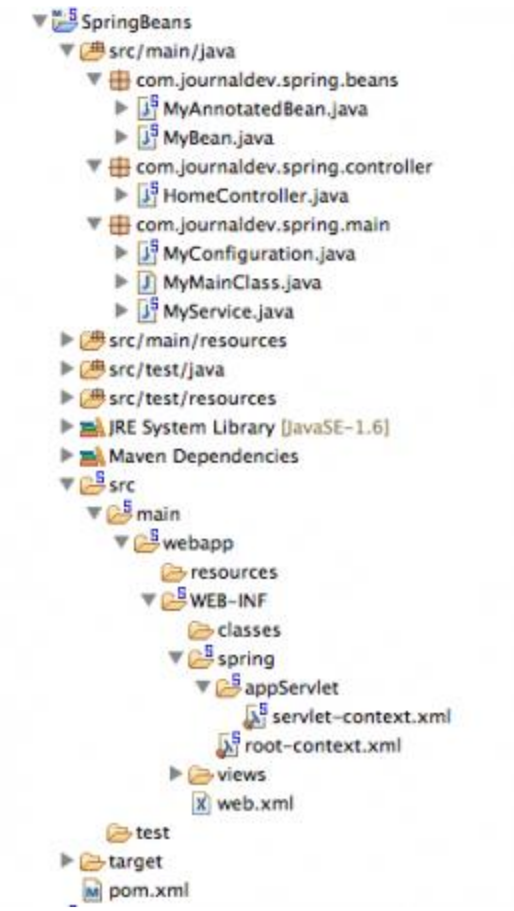
annotations used for java based configuration are @Configuration, @ComponentScan and @Bean.

E. Spring IoC and Bean Example Project

Let’s look at the different aspects of Spring IoC container and Spring Bean configurations with a simple Spring project.

For my example, I am creating Spring MVC project in Spring Tool Suite. If you are new to Spring Tool Suite and Spring MVC, please read Spring MVC Tutorial with Spring Tool Suite.

The final project structure looks like below image.



```
▼ SpringBeans
  ▼ src/main/java
    ▼ com.journaldev.spring.beans
      ▶ MyAnnotatedBean.java
      ▶ MyBean.java
    ▼ com.journaldev.spring.controller
      ▶ HomeController.java
    ▼ com.journaldev.spring.main
      ▶ MyConfiguration.java
      ▶ MyMainClass.java
      ▶ MyService.java
  ▶ src/main/resources
  ▶ src/test/java
  ▶ src/test/resources
  ▶ JRE System Library [JavaSE-1.6]
  ▶ Maven Dependencies
  ▼ src
    ▼ main
      ▼ webapp
          resources
        ▼ WEB-INF
            classes
          ▼ spring
            ▼ appServlet
                servlet-context.xml
              root-context.xml
          ▶ views
            web.xml
      test
  ▶ target
    pom.xml
```

Let’s look at different components one by one.

a). XML Based Bean Configuration

MyBean is a simple Java POJO class.

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

public class MyBean {

    private String name;

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

}
```

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

    <!-- DispatcherServlet Context: defines this servlet's request-
processing infrastructure -->

    <!-- Enables the Spring MVC @Controller programming model -->
    <annotation-driven />

    <!-- Handles HTTP GET requests for /resources/** by efficiently
serving up static resources in the ${webappRoot}/resources directory --
>
    <resources mapping="/resources/**" location="/resources/" />

    <!-- Resolves views selected for rendering by @Controllers to .jsp
resources in the /WEB-INF/views directory -->
    <beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolve
r">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>
    <context:component-scan base-package="com.journaldev.spring" />

    <beans:bean name="myBean"
class="com.journaldev.spring.beans.MyBean" scope="singleton"
></beans:bean>

</beans:beans>
```

Notice that MyBean is configured using *bean* element with scope as singleton.

b). Annotation Based Bean Configuration

```
package com.journaldev.spring.beans;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;
import org.springframework.web.context.WebApplicationContext;

@Service
@Scope(WebApplicationContext.SCOPE_REQUEST)
public class MyAnnotatedBean {

    private int empId;

    public int getEmpId() {
        return empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }

}
```

MyAnnotatedBean is configured using @Service and scope is set to Request.

Controller Class

HomeController class will handle the HTTP requests for the home page of the application. We will inject our Spring beans to this controller class through WebApplicationContext container.

```
package com.journaldev.spring.controller;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.journaldev.spring.beans.MyAnnotatedBean;
import com.journaldev.spring.beans.MyBean;

@Controller
@Scope("request")
public class HomeController {

    private MyBean myBean;

    private MyAnnotatedBean myAnnotatedBean;

    @Autowired
    public void setMyBean(MyBean myBean) {
        this.myBean = myBean;
    }

    @Autowired
    public void setMyAnnotatedBean(MyAnnotatedBean obj) {
        this.myAnnotatedBean = obj;
    }

    /**
     * Simply selects the home view to render by returning its name.
     */
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String home(Locale locale, Model model) {
        System.out.println("MyBean hashcode="+myBean.hashCode());
        System.out.println("MyAnnotatedBean
hashcode="+myAnnotatedBean.hashCode());

        Date date = new Date();
        DateFormat dateFormat =
DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG,
locale);

        String formattedDate = dateFormat.format(date);

        model.addAttribute("serverTime", formattedDate );

        return "home";
    }

}
```

Deployment Descriptor

We need to configure our application for Spring Framework, so that the configuration metadata will get loaded and context will be initialized.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee 
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <!-- The definition of the Root Spring Container shared by all 
Servlets and Filters -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml</param-value>
    </context-param>

    <!-- Creates the Spring Container shared by all Servlets and 
Filters -->
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
    </listener>

    <!-- Processes application requests -->
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring/appServlet/servlet-
context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Almost all the configuration above is boiler-plate code generated by STS tool automatically.

Run the Web Application

Now when you will launch the web application, the home page will get loaded and in the console following logs will be printed when you refresh the page multiple times.

```
MyBean hashcode=118267258
MyAnnotatedBean hashcode=1703899856
MyBean hashcode=118267258
MyAnnotatedBean hashcode=1115599742
MyBean hashcode=118267258
MyAnnotatedBean hashcode=516457106
```

Notice that MyBean is configured to be singleton, so the container is always returning the same instance and hashcode is always same. Similarly for each request, a new instance of MyAnnotatedBean is created with different hashcode.

c). Java Based Bean Configuration

For standalone applications, we can use annotation based as well as xml based configuration. The only requirement is to initialize the context somewhere in the program before we use it.

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

import java.util.Date;

public class MyService {

    public void log(String msg){
        System.out.println(new Date()+"::"+msg);
    }
}
```

MyService is a simple java class with some methods.

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

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@ComponentScan(value="com.journaldev.spring.main")
public class MyConfiguration {

    @Bean
    public MyService getService(){
        return new MyService();
    }
}
```

The annotation based configuration class that will be used to initialize the Spring container.

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

import
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt;

public class MyMainClass {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(
                MyConfiguration.class);
        MyService service = ctx.getBean(MyService.class);

        service.log("Hi");

        MyService newService = ctx.getBean(MyService.class);
        System.out.println("service hashcode="+service.hashCode());
        System.out.println("newService
hashcode="+newService.hashCode());
        ctx.close();
    }

}
```

A simple test program where we are initializing the AnnotationConfigApplicationContext context and then using *getBean()* method to get the instance of MyService.

Notice that I am calling getBean method two times and printing the hashcode. Since there is no scope defined for MyService, it should be singleton and hence hashcode should be the same for both the instances.

When we run the above application, we get following console output confirming our understanding.

```
Sat Dec 28 22:49:18 PST 2013::Hi
service hashcode=678984726
newService hashcode=678984726
```

If you are looking for XML based configuration, just create the Spring XML config file and then initialize the context with following code snippet.

```
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
                "applicationContext.xml");
        MyService app = context.getBean(MyService.class);
```

That's all for the Spring IoC container and Spring Bean Scopes and Configuration details

You can download the source code of "Spring Bean Project" from here.

4. Spring Bean Life Cycle 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.

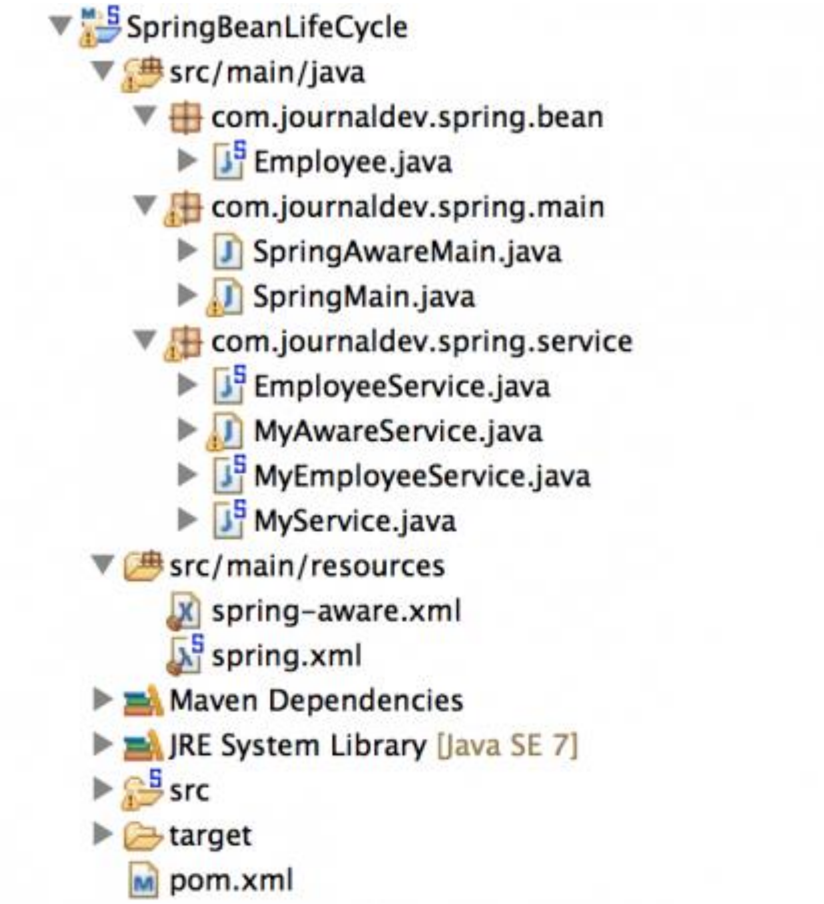
2. Providing **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

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.

A. @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.



```
▼ SpringBeanLifeCycle
  ▼ src/main/java
    ▼ com.journaldev.spring.bean
      ▶ Employee.java
    ▼ com.journaldev.spring.main
      ▶ SpringAwareMain.java
      ▶ SpringMain.java
    ▼ com.journaldev.spring.service
      ▶ EmployeeService.java
      ▶ MyAwareService.java
      ▶ MyEmployeeService.java
      ▶ MyService.java
  ▼ src/main/resources
      spring-aware.xml
      spring.xml
  ▶ Maven Dependencies
  ▶ JRE System Library [Java SE 7]
  ▶ src
  ▶ target
    pom.xml
```

B. 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.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <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.outputEncoding>

        <!-- 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>
```

C. Model Class

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

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

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

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

}
```

D. InitializingBean and Disposable Bean Example

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

```
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");
        }
    }
}
```

E. 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.

```
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");
        }
    }
}
```

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.

F. @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.

```
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.

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

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

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

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

<!-- initializing CommonAnnotationBeanPostProcessor is same as
context:annotation-config -->
<bean
class="org.springframework.context.annotation.CommonAnnotationBeanPostP
rocessor" />
<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.

```
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
ClassPathXmlApplicationContext("spring.xml");

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

        //MyEmployeeService service = ctx.getBean("myEmployeeService",
MyEmployeeService.class);
```

```
        EmployeeService service = ctx.getBean("employeeService",
EmployeeService.class);

        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.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@c1b9
b03: startup date [Tue Apr 01 22:50:50 PDT 2014]; root of context
hierarchy
Apr 01, 2014 10:50:50 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
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.ClassPathXmlApplicationContext
doClose
INFO: Closing
org.springframework.context.support.ClassPathXmlApplicationContext@c1b9
b03: startup date [Tue Apr 01 22:50:50 PDT 2014]; root of context
hierarchy
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.

H. 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.

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

import java.util.Arrays;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanClassLoaderAware;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.context.EnvironmentAware;
import org.springframework.context.ResourceLoaderAware;
import org.springframework.context.annotation.ImportAware;
import org.springframework.core.env.Environment;
import org.springframework.core.io.Resource;
import org.springframework.core.io.ResourceLoader;
import org.springframework.core.type.AnnotationMetadata;

public class MyAwareService implements ApplicationContextAware,
        ApplicationEventPublisherAware, BeanClassLoaderAware,
BeanFactoryAware,
        BeanNameAware, EnvironmentAware, ImportAware,
ResourceLoaderAware {

    @Override
    public void setApplicationContext(ApplicationContext ctx)
            throws BeansException {
        System.out.println("setApplicationContext called");
        System.out.println("setApplicationContext:: Bean Definition
Names="
```

```
                + Arrays.toString(ctx.getBeanDefinitionNames()));
    }

    @Override
    public void setBeanName(String beanName) {
        System.out.println("setBeanName called");
        System.out.println("setBeanName:: Bean Name defined in
context="
                + beanName);
    }

    @Override
    public void setBeanClassLoader(ClassLoader classLoader) {
        System.out.println("setBeanClassLoader called");
        System.out.println("setBeanClassLoader:: ClassLoader Name="
                + classLoader.getClass().getName());
    }

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        System.out.println("setResourceLoader called");
        Resource resource =
resourceLoader.getResource("classpath:spring.xml");
        System.out.println("setResourceLoader:: Resource File Name="
                + resource.getFilename());
    }

    @Override
    public void setImportMetadata(AnnotationMetadata
annotationMetadata) {
        System.out.println("setImportMetadata called");
    }

    @Override
    public void setEnvironment(Environment env) {
        System.out.println("setEnvironment called");
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws
BeansException {
        System.out.println("setBeanFactory called");
        System.out.println("setBeanFactory:: employee bean singleton="
                + beanFactory.isSingleton("employee"));
    }

    @Override
    public void setApplicationEventPublisher(
            ApplicationEventPublisher applicationEventPublisher) {
        System.out.println("setApplicationEventPublisher called");
    }}
```

Spring Bean Configuration File

Very simple spring bean configuration file.

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

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

<bean name="myAwareService"
class="com.journaldev.spring.service.MyAwareService" />

</beans>
```

Spring *Aware Test Program

```
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
ClassPathXmlApplicationContext("spring-aware.xml");

        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.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@60a2
f435: startup date [Tue Apr 01 23:27:05 PDT 2014]; root of context
hierarchy
Apr 01, 2014 11:27:05 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
```

```
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.ClassPathXmlApplicationContext
doClose
INFO: Closing
org.springframework.context.support.ClassPathXmlApplicationContext@60a2
f435: startup date [Tue Apr 01 23:27:05 PDT 2014]; root of context
hierarchy
```

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.

You can download the source code of "Spring Bean Lifecycle Project" from here.

5. Spring Aspect Oriented Programming

Spring Framework is developed on two core concepts – Dependency Injection and Aspect Oriented Programming (AOP). We have already see how Spring Dependency Injection works, today we will look into the core concepts of Aspect Oriented Programming and how we can implement it using Spring Framework.

A. Aspect Oriented Programming Overview

Most of the enterprise applications have some common crosscutting concerns that is applicable for different types of Objects and modules. Some of the common crosscutting concerns are logging, transaction management, data validation etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in Aspect Oriented Programming application modularity is achieved by Aspects and they are configured to cut across different classes.

AOP takes out the direct dependency of crosscutting tasks from classes that we can’t achieve through normal object oriented programming model. For example, we can have a separate class for logging but again the functional classes will have to call these methods to achieve logging across the application.

B. Aspect Oriented Programming Core Concepts

Before we dive into implementation of AOP in Spring Framework, we should understand the core concepts of AOP.

1. **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction

management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using @Aspect annotation.

2. **Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.
3. **Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as Struts2 interceptors or Servlet Filters.
4. **Pointcut**: Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.
5. **Target Object**: They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advices are included based on their configuration.
6. **AOP proxy**: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.
7. **Weaving**: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

C. AOP Advice Types

Based on the execution strategy of advices, they are of following types.

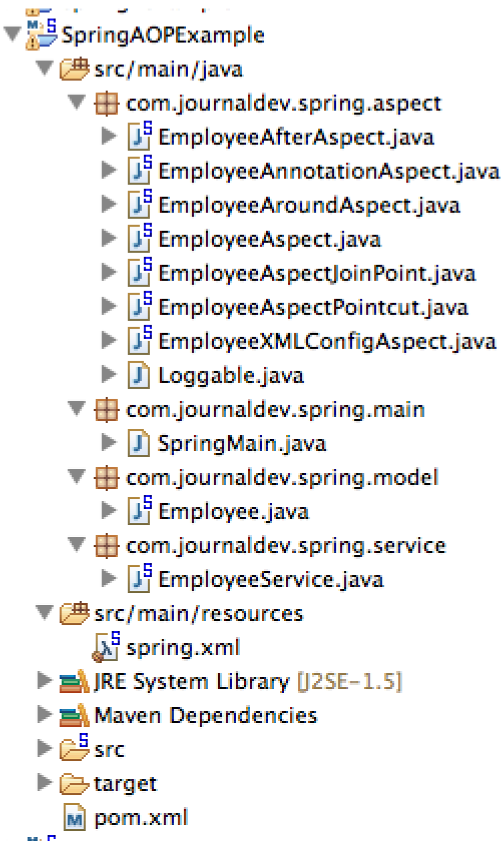
1. **Before Advice**: These advices runs before the execution of join point methods. We can use @Before annotation to mark an advice type as Before advice.
2. **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using @After annotation.

3. **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use @AfterReturning annotation to mark a method as after returning advice.
4. **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use @AfterThrowing annotation for this type of advice.
5. **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use @Around annotation to create around advice methods.

The points mentioned above may sound confusing but when we will look at the implementation of Spring AOP, things will be more clear. Let’s start creating a simple Spring project with AOP implementations. Spring provides support for using AspectJ annotations to create aspects and we will be using that for simplicity. All the above AOP annotations are defined in org.aspectj.lang.annotation package.

Spring Tool Suite provides useful information about the aspects, so I would suggest you to use it. If you are not familiar with STS, I would recommend you to have a look at Spring MVC Tutorial where I have explained how to use it.

Create a new Simple Spring Maven project so that all the Spring Core libraries are included in the pom.xml files and we don’t need to include them explicitly. Our final project will look like below image, we will look into the Spring core components and Aspect implementations in detail.



```
SpringAOPExample
  src/main/java
    com.journaldev.spring.aspect
      EmployeeAfterAspect.java
      EmployeeAnnotationAspect.java
      EmployeeAroundAspect.java
      EmployeeAspect.java
      EmployeeAspectJoinPoint.java
      EmployeeAspectPointcut.java
      EmployeeXMLConfigAspect.java
      Loggable.java
    com.journaldev.spring.main
      SpringMain.java
    com.journaldev.spring.model
      Employee.java
    com.journaldev.spring.service
      EmployeeService.java
  src/main/resources
    spring.xml
  JRE System Library [J2SE-1.5]
  Maven Dependencies
  src
  target
  pom.xml
```

D. Spring AOP AspectJ Dependencies

Spring framework provides AOP support by default but since we are using AspectJ annotations for configuring aspects and advices, we would need to include them in the pom.xml file.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.springframework.samples</groupId>
    <artifactId>SpringAOPExample</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>

        <!-- Generic properties -->
        <java.version>1.6</java.version>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>

        <!-- 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>

        <!-- Test -->
        <junit.version>4.11</junit.version>

        <!-- AspectJ -->
        <aspectj.version>1.7.4</aspectj.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>

        <!-- AspectJ dependencies -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
    </dependencies>
</project>
```

Notice that I have added aspectjrt and aspectjtools dependencies (version 1.7.4) in the project. Also I have updated the Spring framework version to be the latest one as of date i.e 4.0.2.RELEASE.

Model Class

Let's create a simple java bean that we will use for our example with some additional methods.

```
package com.journaldev.spring.model;

import com.journaldev.spring.aspect.Loggable;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

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

    public void throwException(){
        throw new RuntimeException("Dummy Exception");
    }

}
```

Did you noticed that *setName()* method is annotated with Loggable annotation. It is a custom java annotation defined by us in the project. We will look into its usage later on.

Service Class

Let’s create a service class to work with Employee bean.

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

import com.journaldev.spring.model.Employee;

public class EmployeeService {

    private Employee employee;

    public Employee getEmployee(){
        return this.employee;
    }

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

I could have used Spring annotations to configure it as a Spring Component, but we will use XML based configuration in this project. EmployeeService class is very standard and just provides us an access point for Employee beans.

E. Spring Bean Configuration with AOP

If you are using STS, you have option to create “Spring Bean Configuration File” and chose AOP schema namespace but if you are using some other IDE, you can simply add it in the spring bean configuration file.

My project bean configuration file looks like below.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
```

```
<!-- Enable AspectJ style of Spring AOP -->
<aop:aspectj-autoproxy />

<!-- Configure Employee Bean and initialize it -->
<bean name="employee" class="com.journaldev.spring.model.Employee">
    <property name="name" value="Dummy Name"></property>
</bean>

<!-- Configure EmployeeService bean -->
<bean name="employeeService"
class="com.journaldev.spring.service.EmployeeService">
    <property name="employee" ref="employee"></property>
</bean>

<!-- Configure Aspect Beans, without this Aspects advices wont execute
-->
<bean name="employeeAspect"
class="com.journaldev.spring.aspect.EmployeeAspect" />
<bean name="employeeAspectPointcut"
class="com.journaldev.spring.aspect.EmployeeAspectPointcut" />
<bean name="employeeAspectJoinPoint"
class="com.journaldev.spring.aspect.EmployeeAspectJoinPoint" />
<bean name="employeeAfterAspect"
class="com.journaldev.spring.aspect.EmployeeAfterAspect" />
<bean name="employeeAroundAspect"
class="com.journaldev.spring.aspect.EmployeeAroundAspect" />
<bean name="employeeAnnotationAspect"
class="com.journaldev.spring.aspect.EmployeeAnnotationAspect" />

</beans>
```

For using AOP in Spring beans, we need to do following:

1. Declare AOP namespace like xmlns:aop=”http://www.springframework.org/schema/aop”
2. Add aop:aspectj-autoproxy element to enable Spring AspectJ support with auto proxy at runtime
3. Configure Aspect classes as other Spring beans

You can see that I have a lot of aspects defined in the spring bean configuration file, it’s time to look into those one by one.

Before Aspect Example

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAspect {

    @Before("execution(public String getName())")
    public void getNameAdvice(){
        System.out.println("Executing Advice on getName()");
    }

    @Before("execution(* com.journaldev.spring.service.*.get*())")
    public void getAllAdvice(){
        System.out.println("Service method getter called");
    }
}
```

Important points in above aspect class is:

- Aspect classes are required to have @Aspect annotation.
- @Before annotation is used to create Before advice
- The string parameter passed in the @Before annotation is the Pointcut expression
- *getNameAdvice()* advice will execute for any Spring Bean method with signature public String getName(). This is a very important point to remember, if we will create Employee bean using new operator the advices will not be applied. Only when we will use ApplicationContext to get the bean, advices will be applied.
- We can use asterisk (*) as wild card in Pointcut expressions, *getAllAdvice()* will be applied for all the classes in com.journaldev.spring.service package whose name starts with get and doesn’t take any arguments.

We will look these advices in action in a test class after we have looked into all the different types of advices.

Pointcut Methods and Reuse

Sometimes we have to use same Pointcut expression at multiple places, we can create an empty method with @Pointcut annotation and then use it as expression in advices.

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class EmployeeAspectPointcut {

    @Before("getNamePointcut()")
    public void loggingAdvice(){
        System.out.println("Executing loggingAdvice on getName()");
    }

    @Before("getNamePointcut()")
    public void secondAdvice(){
        System.out.println("Executing secondAdvice on getName()");
    }

    @Pointcut("execution(public String getName())")
    public void getNamePointcut(){}

    @Before("allMethodsPointcut()")
    public void allServiceMethodsAdvice(){
        System.out.println("Before executing service method");
    }

    //Pointcut to execute on all the methods of classes in a package
    @Pointcut("within(com.journaldev.spring.service.*)")
    public void allMethodsPointcut(){}

}
```

Above example is very clear, rather than expression we are using method name in the advice annotation argument.

JoinPoint and Advice Arguments

We can use JoinPoint as parameter in the advice methods and using it get the method signature or the target object.

We can use args() expression in the pointcut to be applied to any method that matches the argument pattern. If we use this, then we need to use the same name in the advice method from where argument type is determined. We can use Generic objects also in the advice arguments.

```
package com.journaldev.spring.aspect;

import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAspectJoinPoint {


    @Before("execution(public void
com.journaldev.spring.model..set*(*))")
    public void loggingAdvice(JoinPoint joinPoint){
        System.out.println("Before running loggingAdvice on
method="+joinPoint.toString());

        System.out.println("Agruments Passed=" +
Arrays.toString(joinPoint.getArgs()));

    }

    //Advice arguments, will be applied to bean methods with single
String argument
    @Before("args(name)")
    public void logStringArguments(String name){
        System.out.println("String argument passed="+name);
    }
}
```

After Advice Example

Let’s look at a simple aspect class with example of After, After Throwing and After Returning advices.

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class EmployeeAfterAspect {

    @After("args(name)")
    public void logStringArguments(String name){
        System.out.println("Running After Advice. String argument
passed="+name);
    }
    @AfterThrowing("within(com.journaldev.spring.model.Employee)")
    public void logExceptions(JoinPoint joinPoint){
        System.out.println("Exception thrown in Employee
Method="+joinPoint.toString());
    }

    @AfterReturning(pointcut="execution(* getName())",
returning="returnString")
    public void getNameReturningAdvice(String returnString){
        System.out.println("getNameReturningAdvice executed. Returned
String="+returnString);
    }

}
```

We can use within in pointcut expression to apply advice to all the methods in the class. We can use @AfterReturning advice to get the object returned by the advised method.
We have *throwException()* method in the Employee bean to showcase the use of After Throwing advice.

Spring Around Aspect Example

As explained earlier, we can use Around aspect to cut the method execution before and after. We can use it to control whether the advised method will execute or not. We can also inspect the returned value and change it. This is the most powerful advice and needs to be applied properly.

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class EmployeeAroundAspect {

    @Around("execution(*
com.journaldev.spring.model.Employee.getName())")
    public Object employeeAroundAdvice(ProceedingJoinPoint
proceedingJoinPoint){
        System.out.println("Before invoking getName() method");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("After invoking getName() method. Return
value="+value);
        return value;
    }
}
```

Around advice are always required to have ProceedingJoinPoint as argument and we should use it's proceed() method to invoke the target object advised method. If advised method is returning something, it's advice responsibility to return it to the caller program. For void methods, advice method can return null. Since around advice cut around the advised method, we can control the input and output of the method as well as its execution behavior.

Advice with Custom Annotation Pointcut

If you look at all the above advices pointcut expressions, there are chances that they gets applied to some other beans where it's not intended. For example, someone can define a new spring bean with getName() method and the advices will start getting applied to that even though it was not intended. That's why we should keep the scope of pointcut expression as narrow as possible.

An alternative approach is to create a custom annotation and annotate the methods where we want the advice to be applied. This is the purpose of having Employee *setName()* method annotated with @Loggable annotation.

Spring Framework @Transactional annotation is a great example of this approach for Spring Transaction Management.

```
package com.journaldev.spring.aspect;

public @interface Loggable {

}
```

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAnnotationAspect {
    @Before("@annotation(com.journaldev.spring.aspect.Loggable)")
    public void myAdvice(){
        System.out.println("Executing myAdvice!!");
    }
}
```

myAdvice() method will advice only setName() method. This is a very safe approach and whenever we want to apply the advice on any method, all we need is to annotate it with Loggable annotation.

Spring AOP XML Configuration

I always prefer annotation but we also have option to configure aspects in spring configuration file. For example, let's say we have a class as below.

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

public class EmployeeXMLConfigAspect {

    public Object employeeAroundAdvice(ProceedingJoinPoint
proceedingJoinPoint){
        System.out.println("EmployeeXMLConfigAspect:: Before invoking
getName() method");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("EmployeeXMLConfigAspect:: After invoking
getName() method. Return value="+value);
        return value;
    }
}
```

We can configure it by including following configuration in the Spring Bean config file.

```
<bean name="employeeXMLConfigAspect"
class="com.journaldev.spring.aspect.EmployeeXMLConfigAspect" />

<!-- Spring AOP XML Configuration -->
<aop:config>
    <aop:aspect ref="employeeXMLConfigAspect"
id="employeeXMLConfigAspectID" order="1">
        <aop:pointcut expression="execution(*
com.journaldev.spring.model.Employee.getName())" id="getNamePointcut"/>
```

```
        <aop:around method="employeeAroundAdvice" pointcut-
ref="getNamePointcut" arg-names="proceedingJoinPoint"/>
    </aop:aspect>
</aop:config>
```

AOP xml config elements purpose is clear from their name, so I won’t go into much detail about it.

F. Spring AOP in Action

Let’s have a simple Spring program and see how all these aspects cut through the bean methods.

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

import
org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.service.EmployeeService;

public class SpringMain {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("spring.xml");
        EmployeeService employeeService =
ctx.getBean("employeeService", EmployeeService.class);

        System.out.println(employeeService.getEmployee().getName());

        employeeService.getEmployee().setName("Pankaj");

        employeeService.getEmployee().throwException();

        ctx.close();
    }

}
```

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

```
Mar 20, 2014 8:50:09 PM
org.springframework.context.support.ClassPathXmlApplicationContext
prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@4b9a
f9a9: startup date [Thu Mar 20 20:50:09 PDT 2014]; root of context
hierarchy
Mar 20, 2014 8:50:09 PM
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource
[spring.xml]
Service method getter called
Before executing service method
EmployeeXMLConfigAspect:: Before invoking getName() method
Executing Advice on getName()
Executing loggingAdvice on getName()
Executing secondAdvice on getName()
Before invoking getName() method
After invoking getName() method. Return value=Dummy Name
getNameReturningAdvice executed. Returned String=Dummy Name
EmployeeXMLConfigAspect:: After invoking getName() method. Return
value=Dummy Name
Dummy Name
Service method getter called
Before executing service method
String argument passed=Pankaj
Before running loggingAdvice on method=execution(void
com.journaldev.spring.model.Employee.setName(String))
Agruments Passed=[Pankaj]
Executing myAdvice!!
Running After Advice. String argument passed=Pankaj
Service method getter called
Before executing service method
Exception thrown in Employee Method=execution(void
com.journaldev.spring.model.Employee.throwException())
Exception in thread "main" java.lang.RuntimeException: Dummy Exception
    at
com.journaldev.spring.model.Employee.throwException(Employee.java:19)
    at
com.journaldev.spring.model.Employee$$FastClassBySpringCGLIB$$da2dc051.
invoke(<generated>)
    at
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204
)
    at
org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.i
nvokeJoinpoint(CglibAopProxy.java:711)
```

```
    at
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
flectiveMethodInvocation.java:157)
    at
org.springframework.aop.aspectj.AspectJAfterThrowingAdvice.invoke(Aspec
tJAfterThrowingAdvice.java:58)
    at
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
flectiveMethodInvocation.java:179)
    at
org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(
ExposeInvocationInterceptor.java:92)
    at
org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Re
flectiveMethodInvocation.java:179)
    at
org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedIntercept
or.intercept(CglibAopProxy.java:644)
    at
com.journaldev.spring.model.Employee$$EnhancerBySpringCGLIB$$3f881964.t
hrowException(<generated>)
    at com.journaldev.spring.main.SpringMain.main(SpringMain.java:17)
```

You can see that advices are getting executed one by one based on their pointcut configurations. You should configure them one by one to avoid confusion.

That's all for Spring AOP Tutorial, I hope you learned the basics of AOP with Spring and can learn more from examples.

You can download the source code of "Spring AOP Project" from here.

Copyright Notice

Although the author and publisher have made every effort to ensure that the information in this book was correct at press time, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause. Please report any errors by sending an email to Pankaj.0323@gmail.com

References

1. http://www.journaldev.com/2394/dependency-injection-design-pattern-in-java-example-tutorial
2. http://www.journaldev.com/2410/spring-dependency-injection-example-with-annotations-and-xml-configuration
3. http://www.journaldev.com/2461/spring-ioc-container-and-spring-bean-example-tutorial
4. http://www.journaldev.com/2637/spring-bean-life-cycle-methods-initializingbean-disposablebean-postconstruct-predestroy-aware-interfaces
5. http://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations-xml-configuration