**Spring Framework using AOP**

* Create a maven project name DemoApp. QuickStart with internal type.
* Next make sure Spring context and junit are already in your pom.xml file.
* Pom.xml file should look like this:



* Then create another package in main/java called com.springaop.DemoApp.bean and within this new package create another bean class called **Product**.
* Here we are going to give some details like name, price, brand, and stock. Then do a default constructor, getter/setter and toString method.
* So far, Product.java looks like this: Note that there is one more } at the end to complete this code.

A screenshot of a computer code

Description automatically generated

* Next create an xml file, can name it context.xml and add it to our main/java
* Now in xml file, begin by creating a bean for Product object and its property values. As of now, looks like this:

A computer screen shot of a computer code

Description automatically generated

* Next go to Product class and above toString method, create a constructor with parameters and pass some business information like delivery location and then give an if/else statement depending on the outcome.
* The new Constructor with business logic looks like this:

A screenshot of a computer

Description automatically generated

* Now go to App.java and create a print statement. Then create a IOC Container that is AppplicationContext and pass our xml file so its able to read the content that is context.xml.
* Then create the object of Product, use method context.getBean and pass the reference id name that we given in xml which is pRef.
* Then create a method using product object and productPurchase which is the name of the constructor with parameters we made in Product class. And give those values to the properties we listed inside the constructors parameters.
* App.java looks like this so far:

A screenshot of a computer program

Description automatically generated

* Output from this so far looks like this:

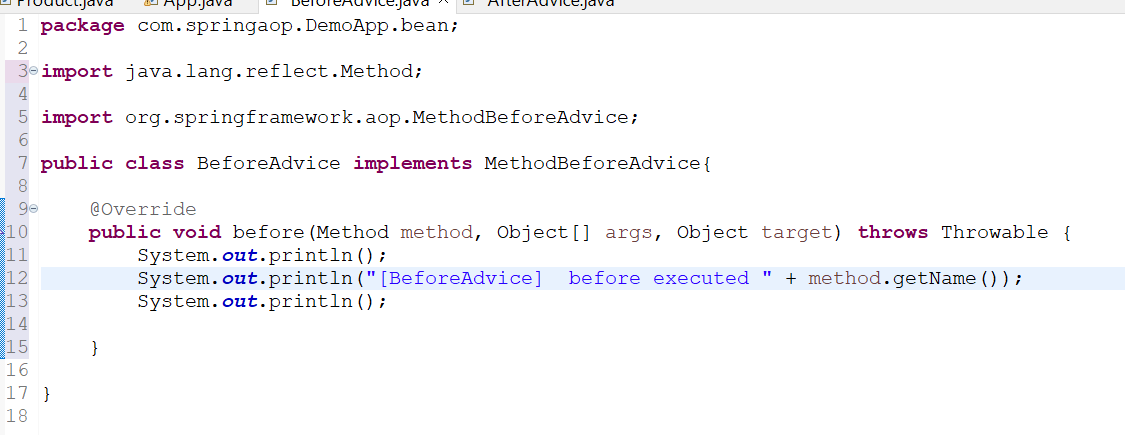
A close-up of a computer screen

Description automatically generated

* But in the future, this business logic should be in a model class, separate from this, it is much easier/more organized.
* Next in bean package, create a class called **BeforeAdvice** and here we are going to implements MethodBeforeAdvice which is a interface. And this **interface extends BeforeAdvice class**.
* This interface contains some **contracts**. One of them is this **before contract**

void before(Method method, Object[] args, @Nullable Object target) throws Throwable;

* We are going to **implement this before function** inside our BeforeAdvice implementation class.
* Now go back to BeforeAdvice.java and hover over the class name and add **unimplemented methods**.
* We have **one unimplemented method** which is the method from the interface that I listed above in red. The **name of this method is before**.
* Now create another class called AfterAdvice in the bean package. In this class you are going to **implements AfterReturningAdvice** This is apart of your **Spring AOP**.
* You will have one unimplemented method from interface, do the same as before. This method will be called **afterReturning** and within this method, create a print statement that shows name of the method by using method.getName(). Copy these print statements and add them in to BeforeAdvice and make some changes. If there are some empty print statements I think its because they add spaces.
* As of now BeforeAdvice looks like:



* As of now, AfterAdvice looks like:

A screenshot of a computer code

Description automatically generated

* Now go to context.xml and lets create **two more beans** and their properties/values. We will come back to this in a moment and show the xml page when finished.
* **Note**: Remember that id is a reference name you give to it so it makes that bean/object you creating unique. And the class name is the package name plus the class name at the end.
* Now go to your app.java and lets work on ProxyFactoryBean.
* **ProxyfactoryBean:** A **proxy factory bean is a JavaBean** that creates an AOP proxy that wraps a target object. It is used to apply interceptor logic to an existing target bean. ProxyFactoryBean is a FactoryBean that builds an AOP proxy based on beans in a Spring BeanFactory.
* Now go back to context.xml and **create a bean for proxy**. For the class, you can look up in your dependencies package and find the package that says ProxyfactoryBean.class and copy that package name.
* Now this **ProxyfactoryBean** takes **interceptor names**, and these names are **interceptorNames and targetName**. These are properties so we can use the property tag in xml
* Name we will give target, ref is the same as product reference which is pRef. And the interceptor names. Our BeforeAdvice and AfterAdvice are our interceptor classes and these classes can intercept and apply the Pre-Processing and Post-Processing logic.
* So another **property after the ref** is the interceptorNames, this is a **String array** so we need to give values in a list tag. There are two values, which is the **befAdv and aftAdv**
* And then we need to divide different phases of business logic into their respective advice Spring AOP.
* So as of now, our xml looks like this:

A screenshot of a computer code

Description automatically generated

* Now go to Product.java and below stock property in the top of the class, give a **Boolean value to canBuy** and this is going to be a **validating** **attribute**. Then you need to generate the getter/setters for canBuy. Right click a the end of the last setter to apply this.
* Now in Product.java remove the validate the stock | Pre-Processing statements from step 1, not step 2 in the if condition. But not the actual if condition yet, just the statements within the if.
* Now go to **BeforeAdvice** and below the print statements create a if statement. **Within the if statement, we are going to write logic** for if the method from which the request is coming, its name equals **productPurchase** then lets work on functionality of Pre-Processing.
* After the **method.getName**() now we are going to work on the **validation** part. We want to validate the stock here. And in the Object Target that you can see a the top of the page. What we are setting in Target object is product reference which is **Product object**.
* So this target is nothing but a product object, we can just cast this to product.
* **Note**: In xml, in the property name target, we are setting a reference or ‘ref’ as the complete product object. Target ref is pRef and id for Product bean is pRef, so this target is calling all of the properties and its values from that specific product object we created.
* Now go back to BeforeAdvice. From our Product we can get the stock.
* Create a if statement, and in the if statement if stock is less than 0. We will set the value of **canBuy to false** meaning that we can not buy the product. Print a statement saying out of stock and that you are not able to by.
* And in the else statement, meaning if its greater than 0, then you are able to buy.
* Basically shifting the before logic from Product to here which is on BeforeAdvice. So here is where the **Pre-Processing logic goes**.
* **As of now, BeforeAdvice looks like this:**

A screenshot of a computer

Description automatically generated

* Now go to AfterAdvice, and her we will be talking about the notification part. This is where the **Post-Processing** will go.
* Here we are going to right a if statement that says if method name that’s called = productPurchase than print the method name. Then lets print a statement for notifying in action. And then to get the product object from the target.
* We will be saving the **target to the product object reference.**
* We can pass values through **Object[] args** that are not part of the product bean. So we can pass email information through args[2] meaning in app.java our 2 index is email and it will pass that email.
* Price is apart of product only so we can just call the price in the print statements from product object/bean directly. Brand is also apart of Product. Delivery location is not, so we can pass it buy Object[] Args and pass the 1st index.
* Now since we moved Post-Processing and Pre-Processing logic from main class into their own classes like how it should be. Lets remove that logic from Product.java And then just leave the core business logic which is the 2nd step here. Since we also defined stocks in pre-processing, here in the if statement instead of checking to see if stock is greater than 0, lets just use if we can buy Boolean value.
* And in app.java change the getBean from product bean only which was pRef to now productProxy.
* So now AfterAdvice looks like this:

A screenshot of a computer

Description automatically generated

* Product.java looks like:

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

* App.java looks like this:

A screenshot of a computer program

Description automatically generated

* **Run this and output is:**

A white screen with text

Description automatically generated with medium confidence

* In app.java we are just calling to the productProxy. Once we call the productProxy reference in xml which is pRef. It will give us the product object as well as the properties or the inceptor names from beforeAdvice and afterAdvice.
* In app.java the moment we call productPurchase, it is still apart of Product bean. But the moment the productPurchase method is initiated, your BeforeAdvice method intercepts the call in Product.java where the canBuy is in the if statement and tries to validate the stock which is in the BeforeAdvice logic.
* Once BeforeAdvice is done, your actual productPurchase method is being called, which is doing our core business logic which is step 2. This is located In Product.java. And then AfterAdvice is getting triggered which is basically notifying the action of what happened.