Lesson 02 Demo 01

Implementing Before Advice and After Returning Advice

Objective: To understand Aspect-Oriented Programming in Spring framework by implementing before advice and after returning advice methods

Tool required: Eclipse IDE

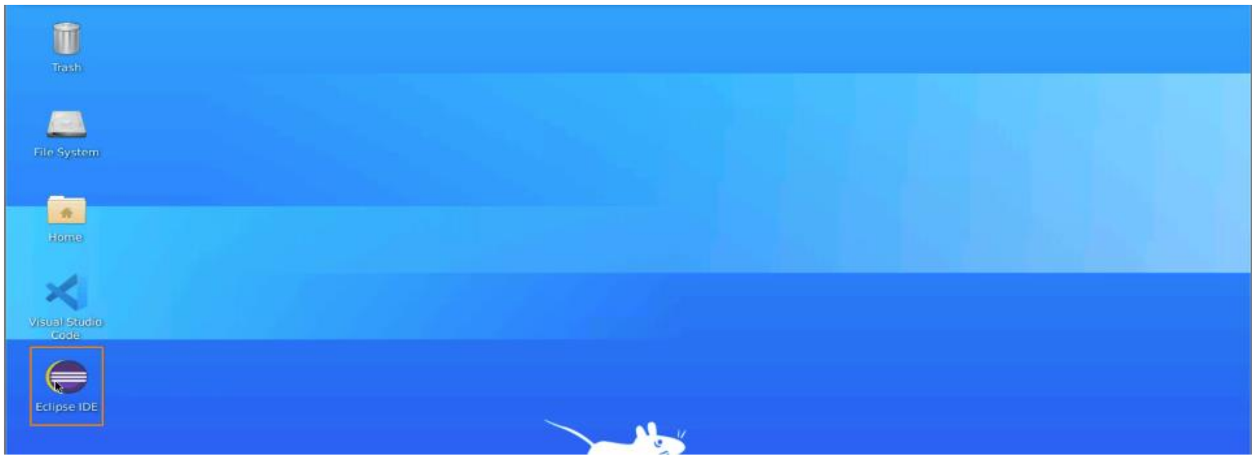
Prerequisites: None

Steps to be followed:

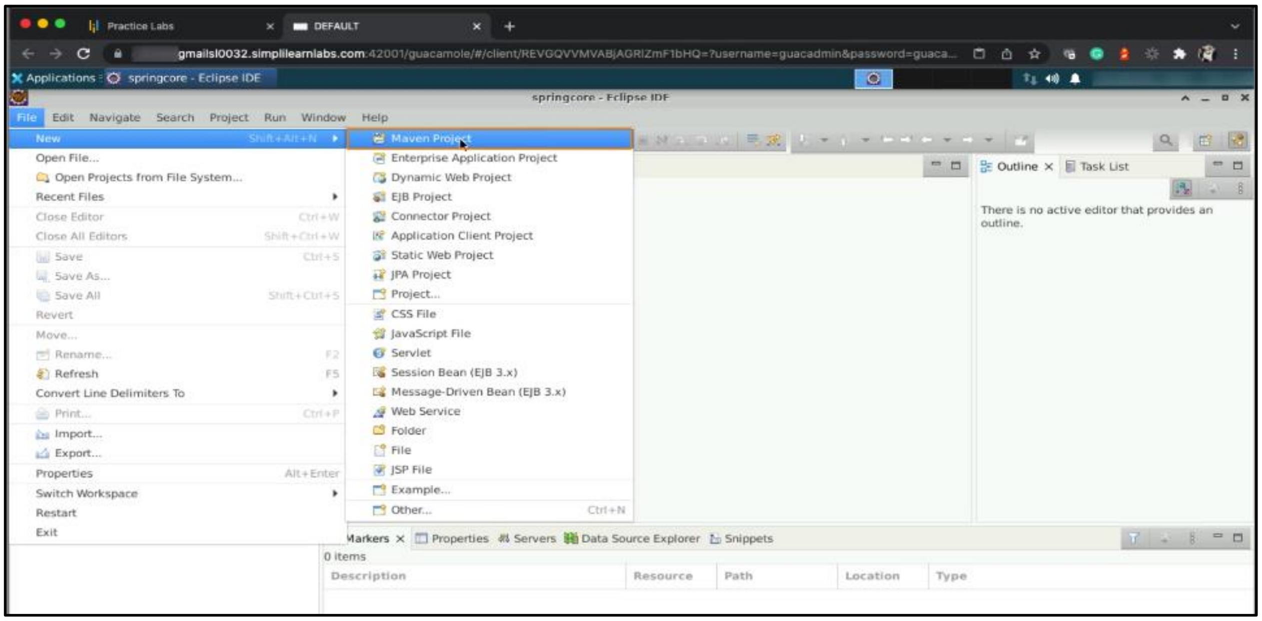
1. Creating a Maven project
2. Configuring the pom.xml file
3. Creating a bean class
4. Implementing the configuration file
5. Adding business logic methods
6. Creating before and after returning advice
7. Configuring both the advice in the XML file
8. Dividing the business logic into before and after returning advice

Step 1: Creating a Maven project

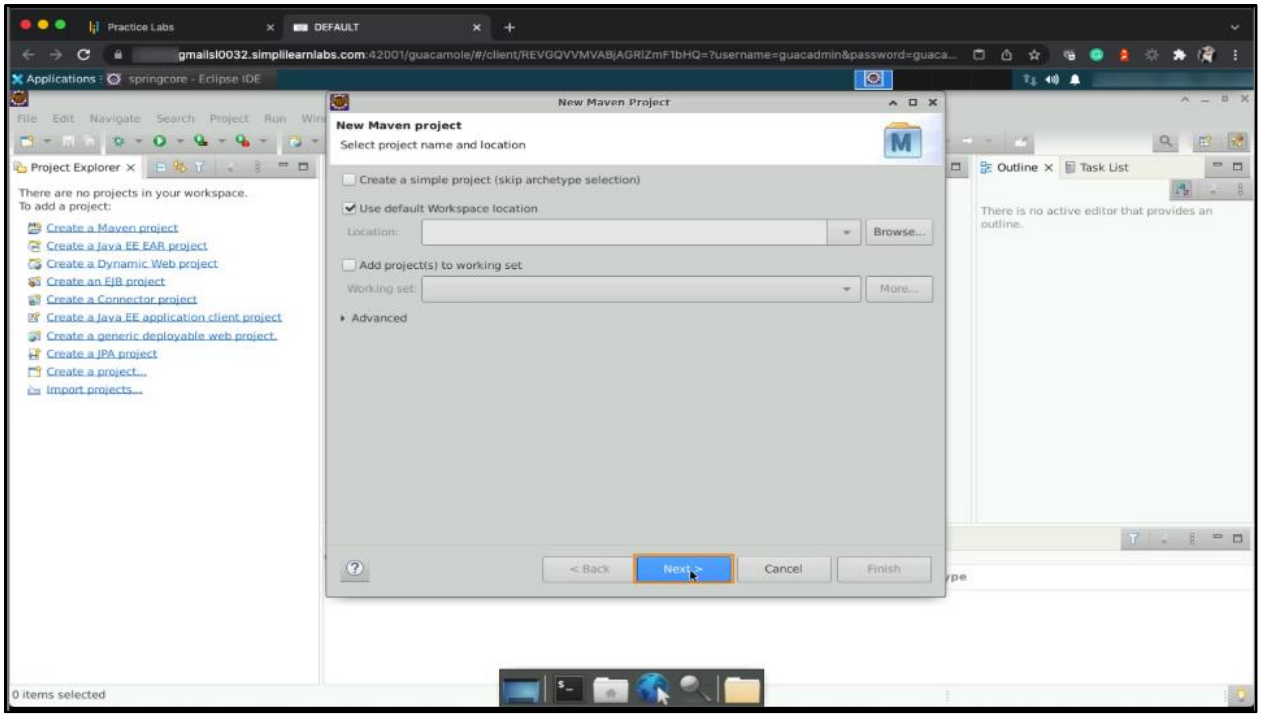
1.1 Open **Eclipse IDE**



1.2 Click on **File** in the menu bar, select **New,** and choose **Maven Project**



1.3 Create a workspace location, which will be the same as the selected workspace for your Eclipse, and click **Next**


New Maven Project

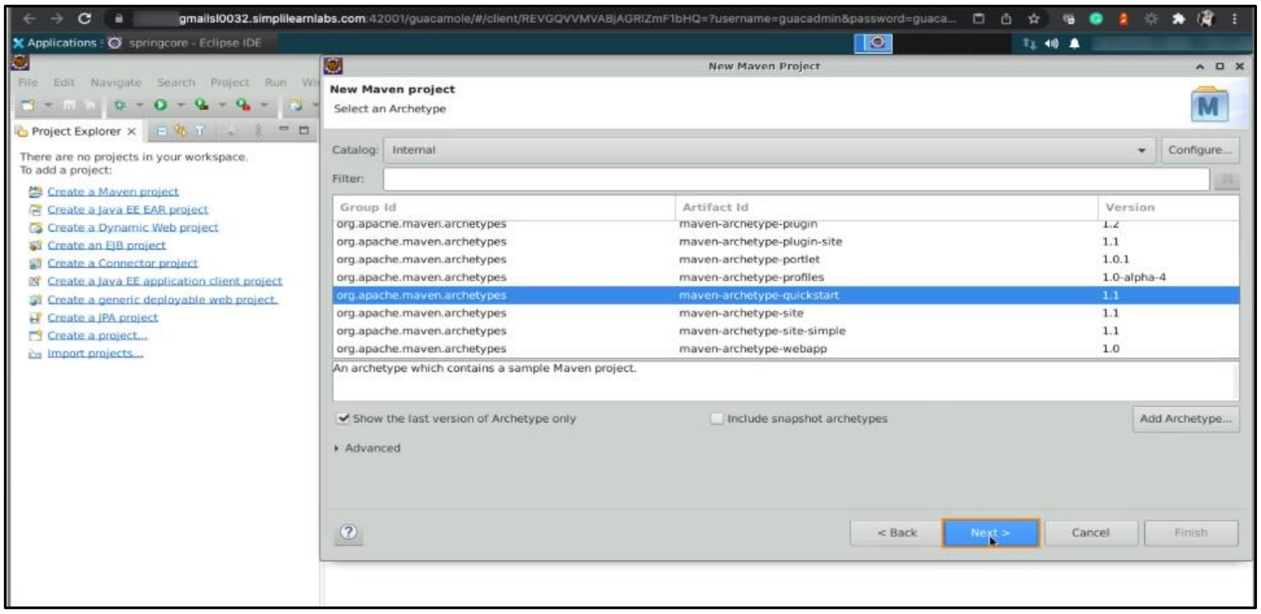
New Maven project

Select project name and location

- [ ] Create a simple project (skip archetype selection)
- [✓] Use default Workspace location
- Location:
- [ ] Add project(s) to working set
- Working set:

Browse...
More...
Advanced
< Back
Next >
Cancel
Finish
Project Explorer
There are no projects in your workspace.
To add a project:
Create a Maven project
Create a Java EE EAR project
Create a Dynamic Web project
Create an EJB project
Create a Connector project
Create a Java EE application client project
Create a generic deployable web project.
Create a JPA project
Create a project...
Import projects...
Outline
Task List
There is no active editor that provides an outline.
0 items selected

1.4 Select the **maven-archetype-quickstart** from the **Internal** catalog and click **Next**



New Maven project
Select an Archetype

There are no projects in your workspace.
To add a project:

Catalog: Internal

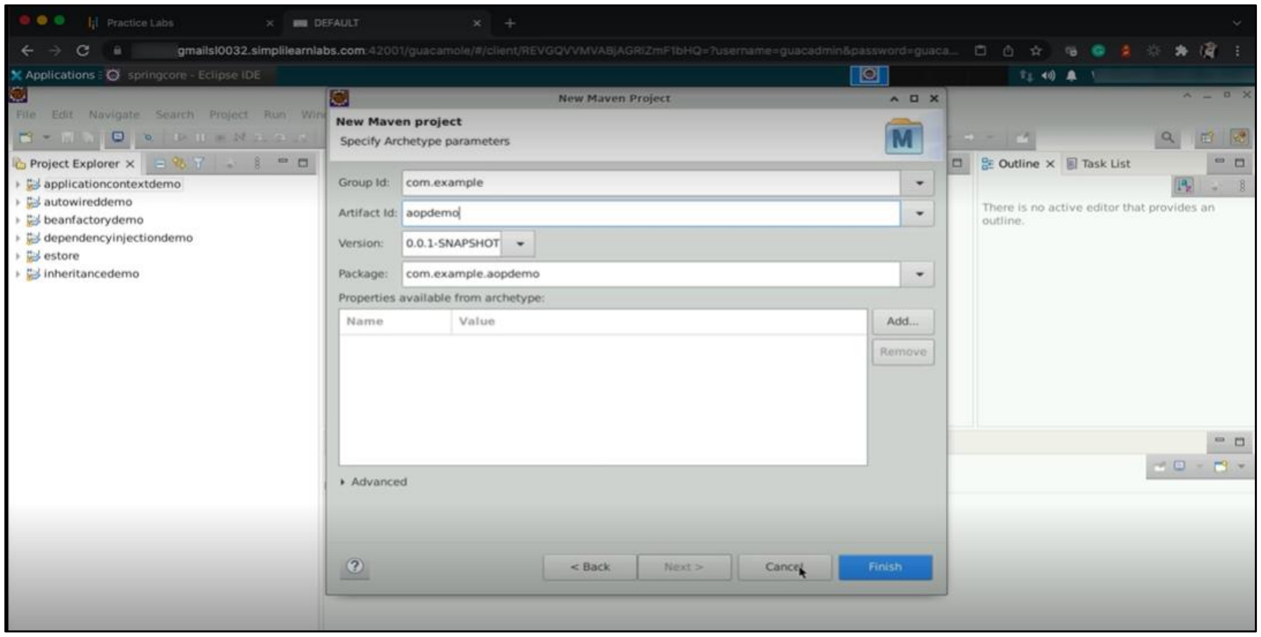
Filter:

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1
org.apache.maven.archetypes	maven-archetype-portlet	1.0.1
org.apache.maven.archetypes	maven-archetype-profiles	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1
org.apache.maven.archetypes	maven-archetype-site	1.1
org.apache.maven.archetypes	maven-archetype-site-simple	1.1
org.apache.maven.archetypes	maven-archetype-webapp	1.0

An archetype which contains a sample Maven project.

[✓] Show the last version of Archetype only
[ ] Include snapshot archetypes

1.5 Provide the **Group Id**, which is typically the company's domain name in reverse order, and the **Artifact Id** as **aopdemo**. Now, click **Finish**



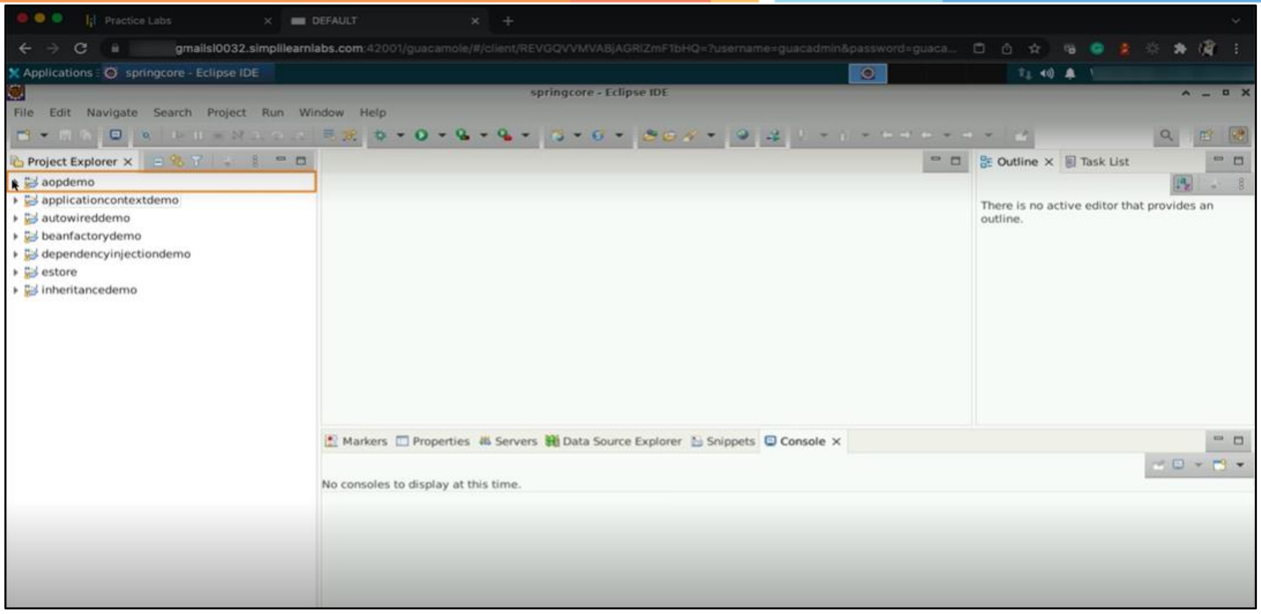
New Maven project
Specify Archetype parameters

Group Id: com.example
Artifact Id: aopdemo
Version: 0.0.1-SNAPSHOT
Package: com.example.aopdemo

Properties available from archetype:

Name	Value

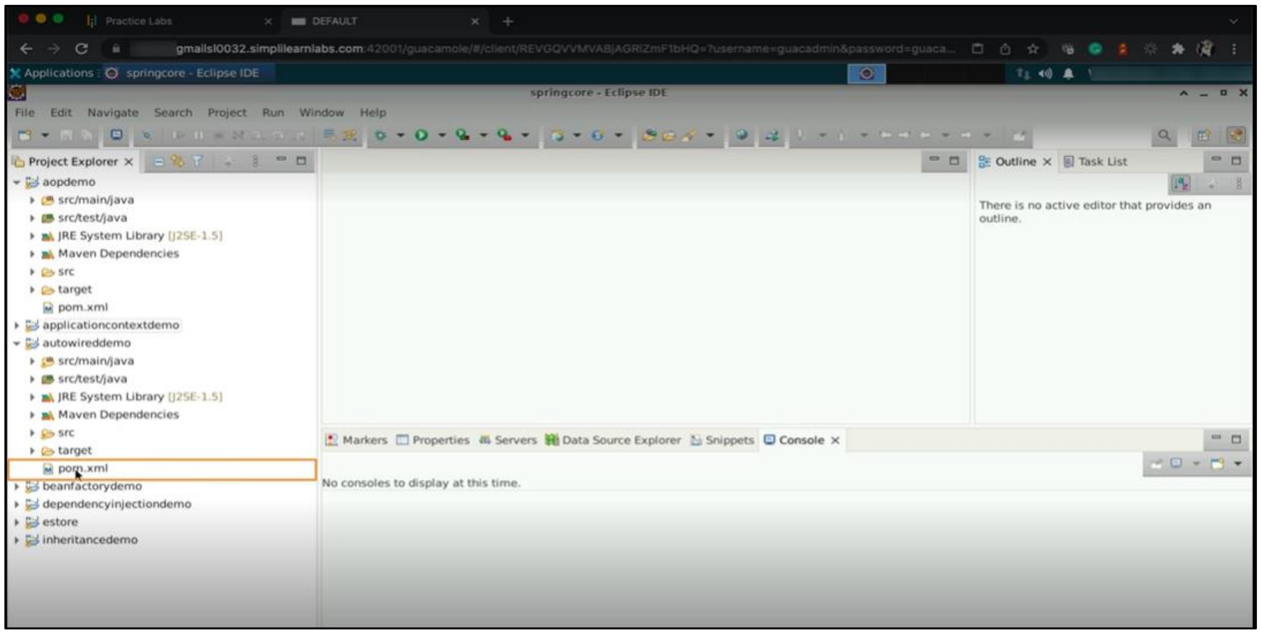
There is no active editor that provides an outline.


Project Explorer
aopdemo
applicationcontextdemo
autowireddemo
beanfactorydemo
dependencyinjectiondemo
estore
inheritancedemo
Outline
Task List
There is no active editor that provides an outline.
Markers Properties Servers Data Source Explorer Snippets Console
No consoles to display at this time.

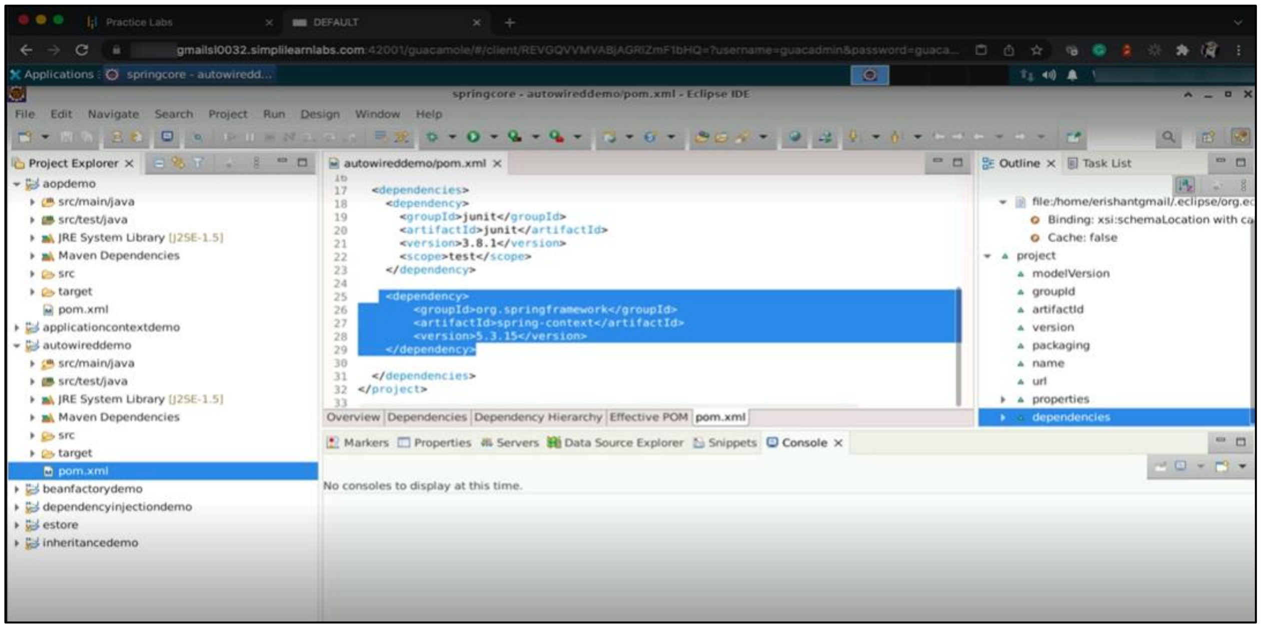
In **Project Explorer**, you will see the newly created **aopdemo** Maven project.

Step 2: Configuring the pom.xml file

2.1 Now we must configure **pom.xml** to define the dependencies. For this, you may copy the dependencies from the previously created **autowireddemo** object. You may also refer to the previous demo for configuring the spring core in the Java project.



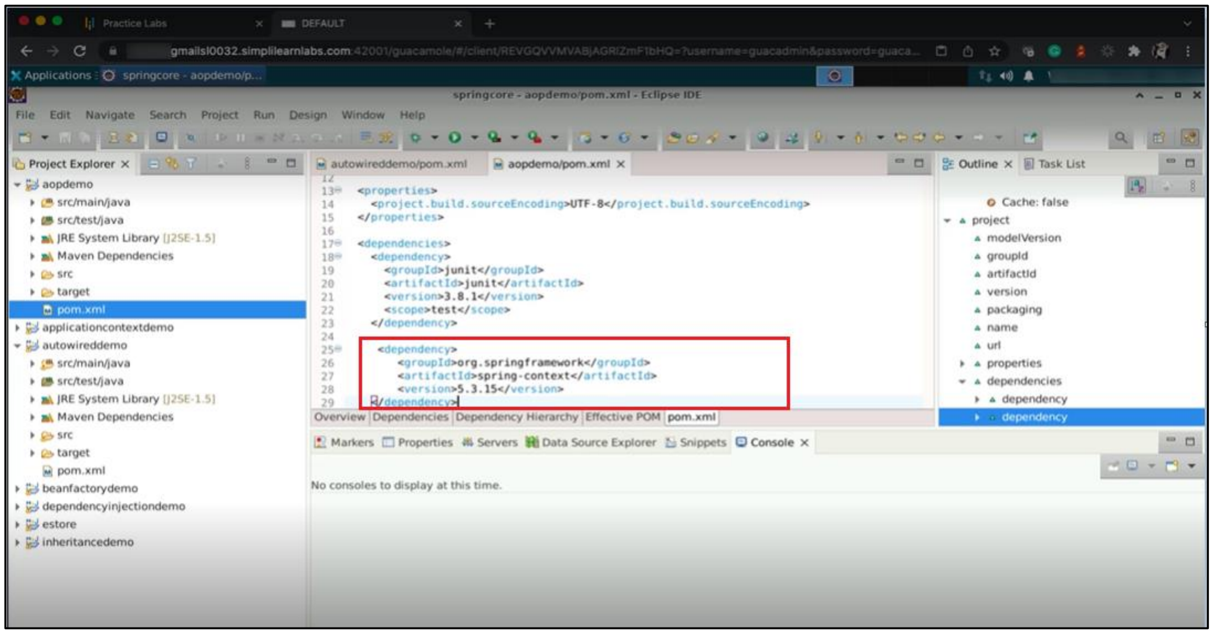
2.2 Copy the dependencies for the Spring Core to configure the pom.xml file for the **aopdemo** object



```
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.15</version>
    </dependency>

  </dependencies>
</project>
```

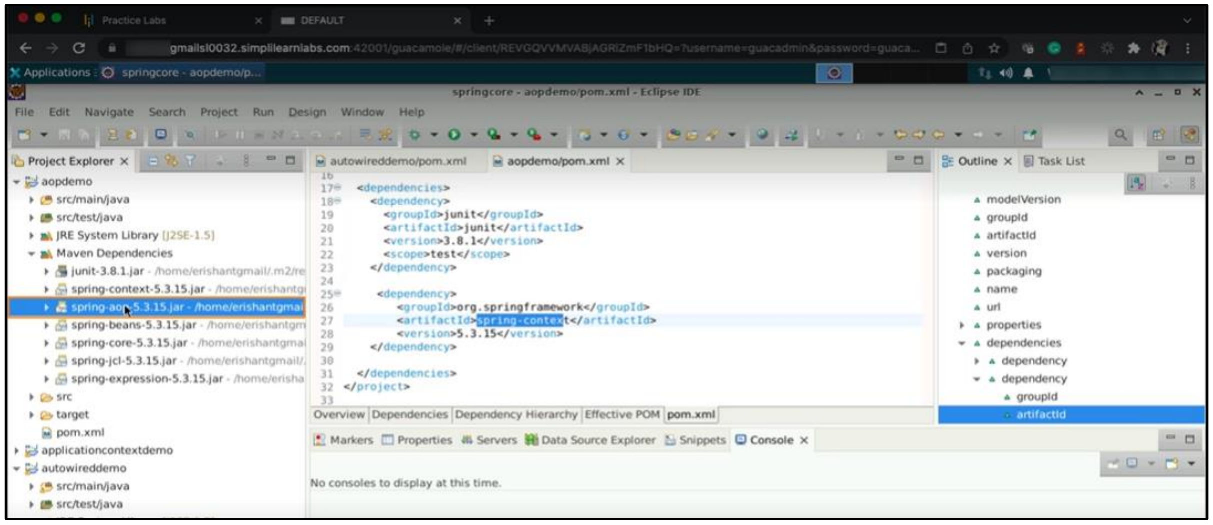
No consoles to display at this time.

2.3 Add the Spring Core dependencies in the pom.xml file of the **aopdemo** object, under the **<dependencies>** section



```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

     <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.15</version>
    </dependency>
```

No consoles to display at this time.



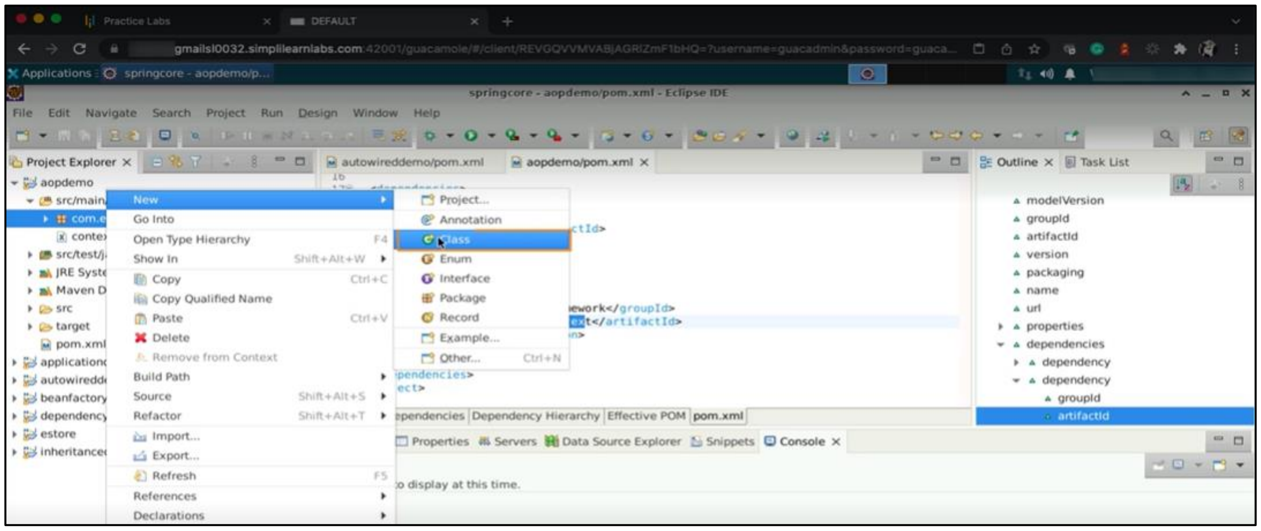
```
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>

       <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>5.3.15</version>
      </dependency>

    </dependencies>
</project>
```

No consoles to display at this time.

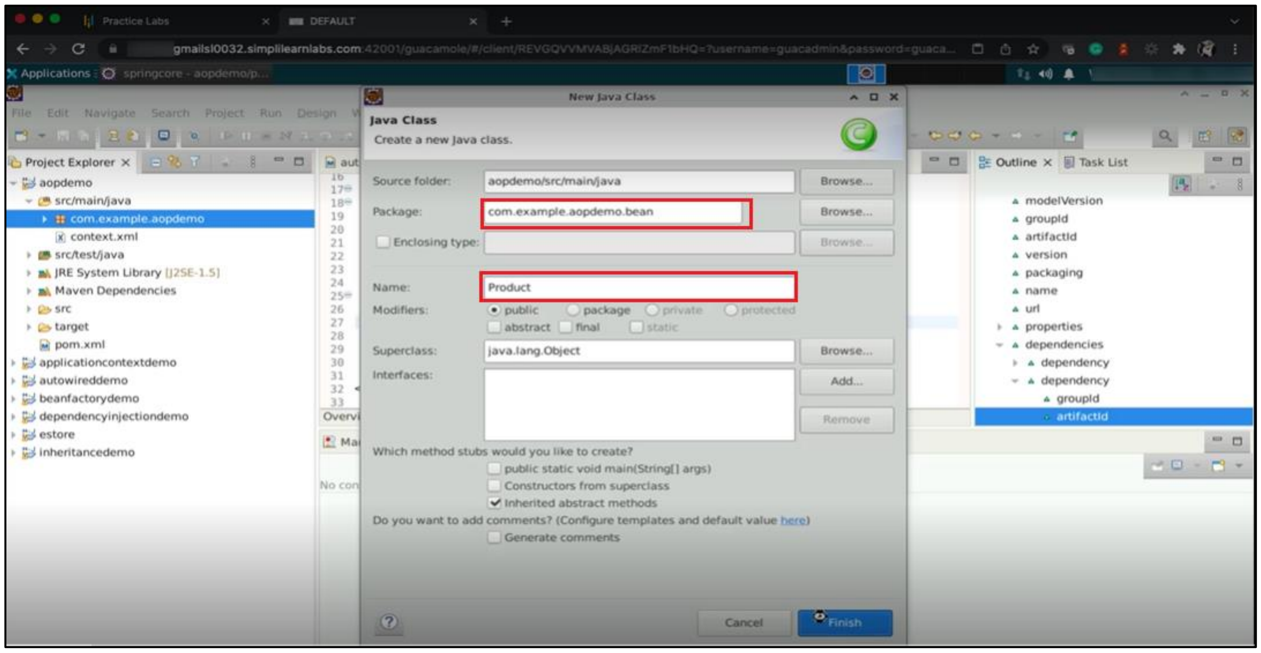
Now your Spring Core dependencies are added to your project. With the **spring-context** artifact Id, you can notice that the required AOP jar files are automatically added under the Maven dependencies.

Step 3: Creating a bean class

3.1 In Eclipse, open the **aopdemo** Java package where you want to create the bean class. Right-click **com.example.aopdemo** package under the **src/main/java folder** and select **New > Class**



3.2 Give the class a name, such as **Product**, and click **Finish.** This class goes under the bean package.

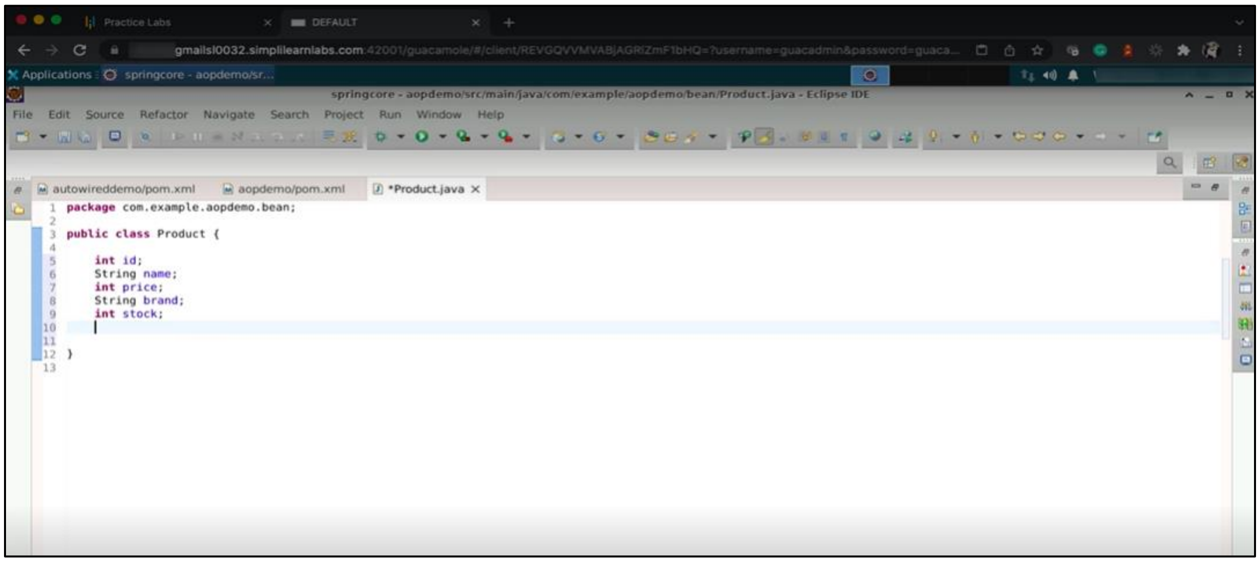

New Java Class

Java Class
Create a new Java class.

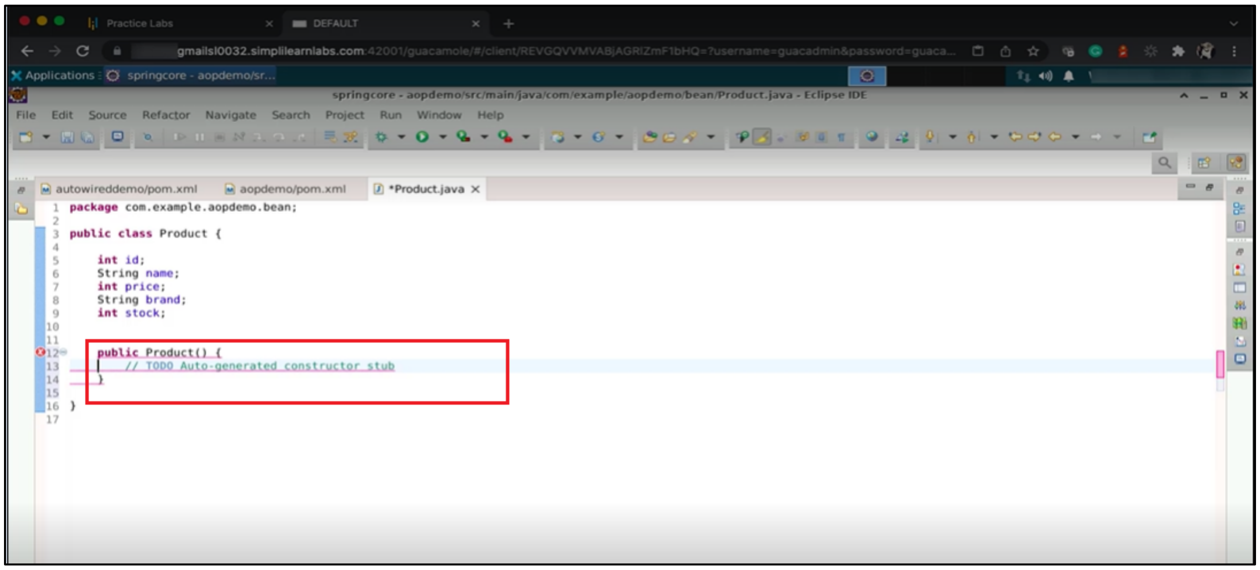
Source folder: aopdemo/src/main/java
Package: com.example.aopdemo.bean
[ ] Enclosing type:
Name: Product
Modifiers: [✓ public] package private protected
[ ] abstract [ ] final [ ] static
Superclass: java.lang.Object
Interfaces:

Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

3.3 In the **Product** class, define attributes, such as **id**, **name**, **price**, **brand,** and **stock**

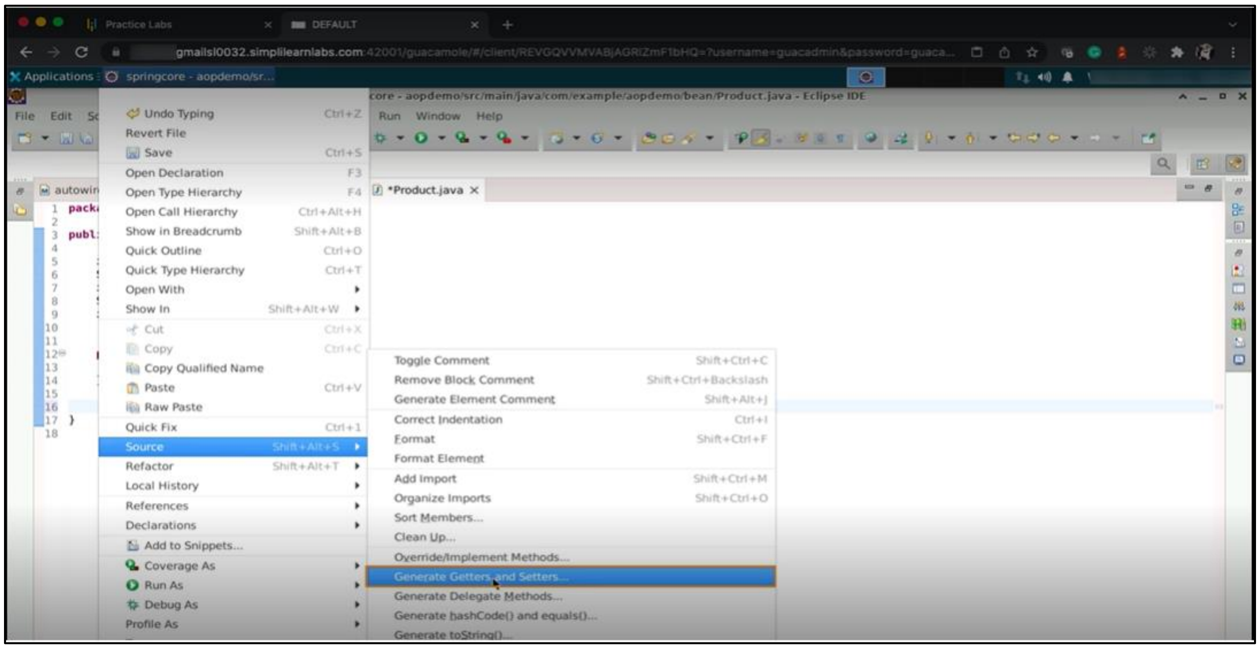

```
package com.example.aopdemo.bean;

public class Product {

    int id;
    String name;
    int price;
    String brand;
    int stock;


}
```

3.4 Create a default constructor for the class **Product**, which will initialize the attributes of the class with default values



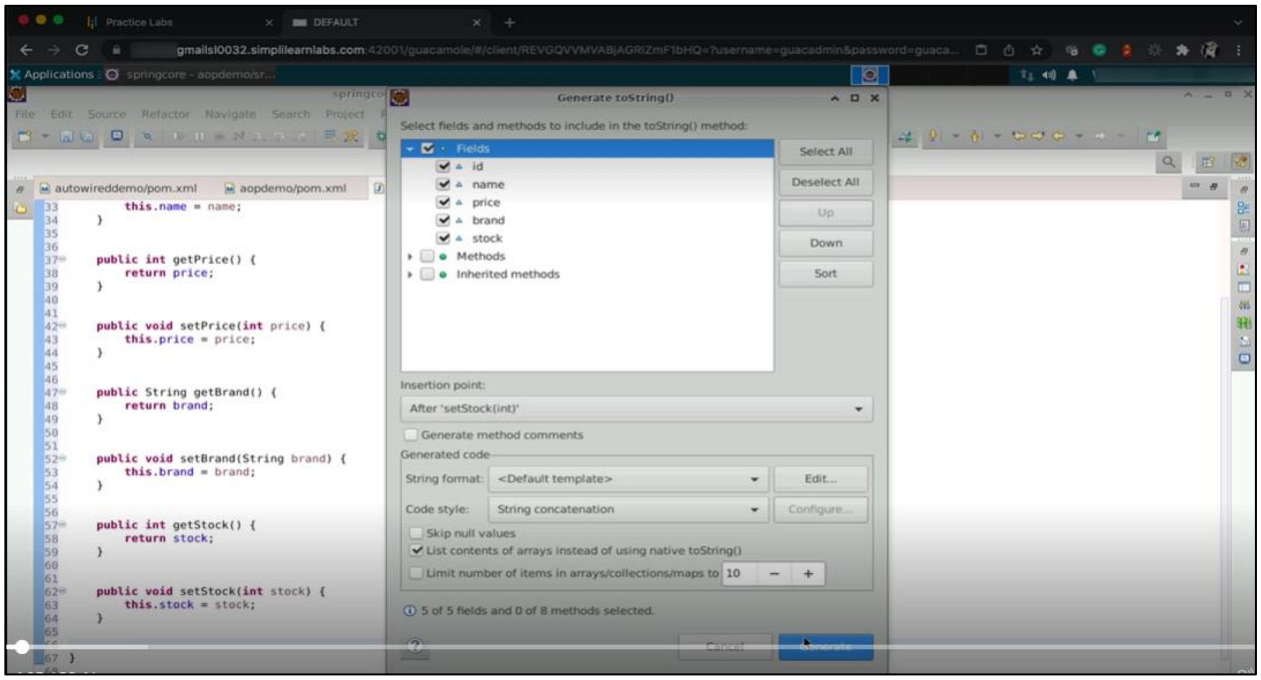
```java
package com.example.aopdemo.bean;

public class Product {

    int id;
    String name;
    int price;
    String brand;
    int stock;


    public Product() {
        // TODO Auto-generated constructor stub
    }

}
```

3.5 Generate getters and setters for the attributes. Right-click and select **Source > Generate Getters and Setters.** Ensure to set all the attributes as **public** for the access modifiers



Note: The getter method is added to return existing values of the class attributes, while the setter is used to set or update any value for the attributes.

3.6 Lastly, generate **toString()** to display data in the object. For that right-click anywhere on the window and select **Source > Generate toString()**



```
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }

}
```

Select fields and methods to include in the toString() method:

- ✓ Fields
  - ✓ id
  - ✓ name
  - ✓ price
  - ✓ brand
  - ✓ stock
- Methods
- Inherited methods

Insertion point: After 'setStock(int)'

Generate method comments

Generated code

String format: <Default template>

Code style: String concatenation

Skip null values

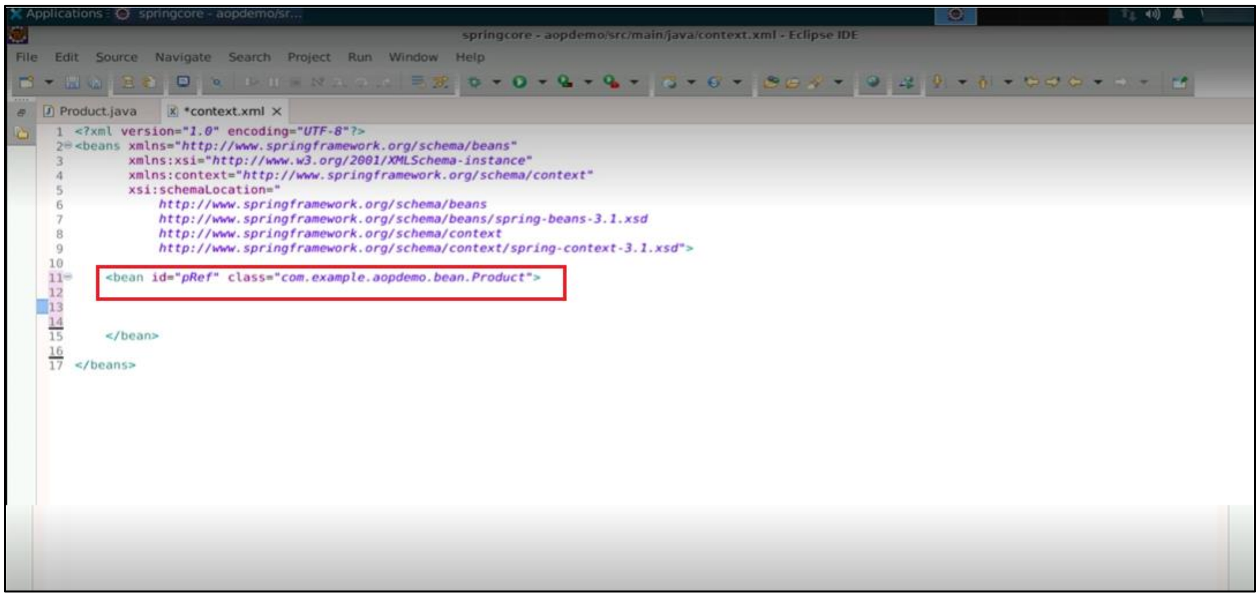
✓ List contents of arrays instead of using native toString()

Limit number of items in arrays/collections/maps to 10

5 of 5 fields and 0 of 8 methods selected.

Step 4: Implementing the configuration file

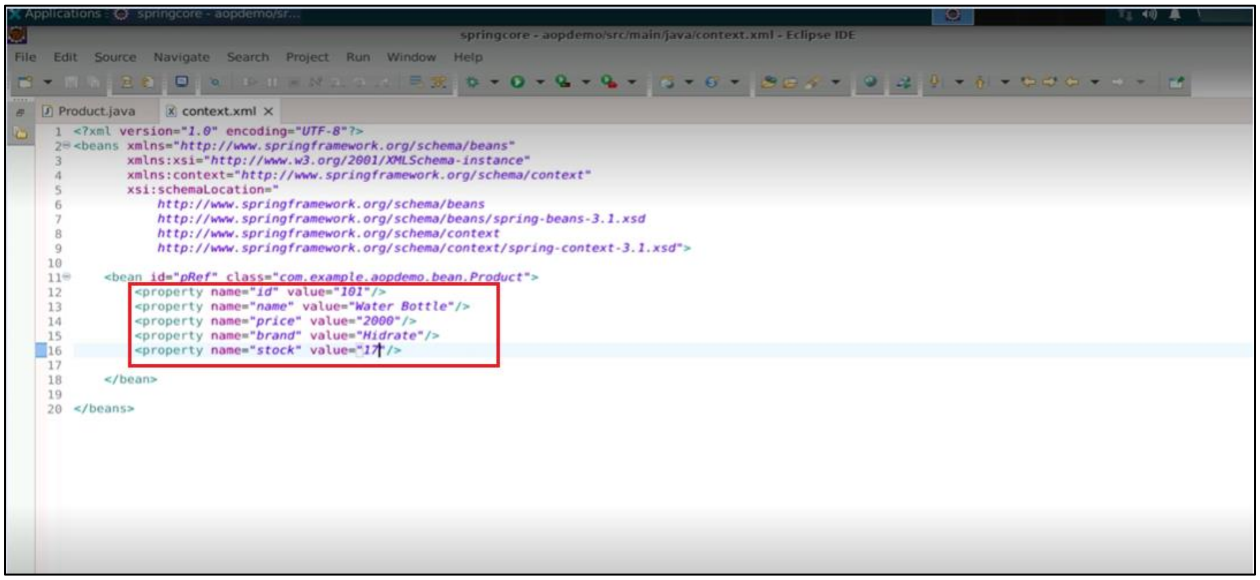
4.1 After configuring the dependencies for the XML file, open the context.xml file and set the bean id and class as **pref** and **com.example.aopdemo.bean.Product**



springcore - aopdemo/src/main/java/context.xml - Eclipse IDE

```
<?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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <bean id="pRef" class="com.example.aopdemo.bean.Product">



    </bean>

</beans>
```

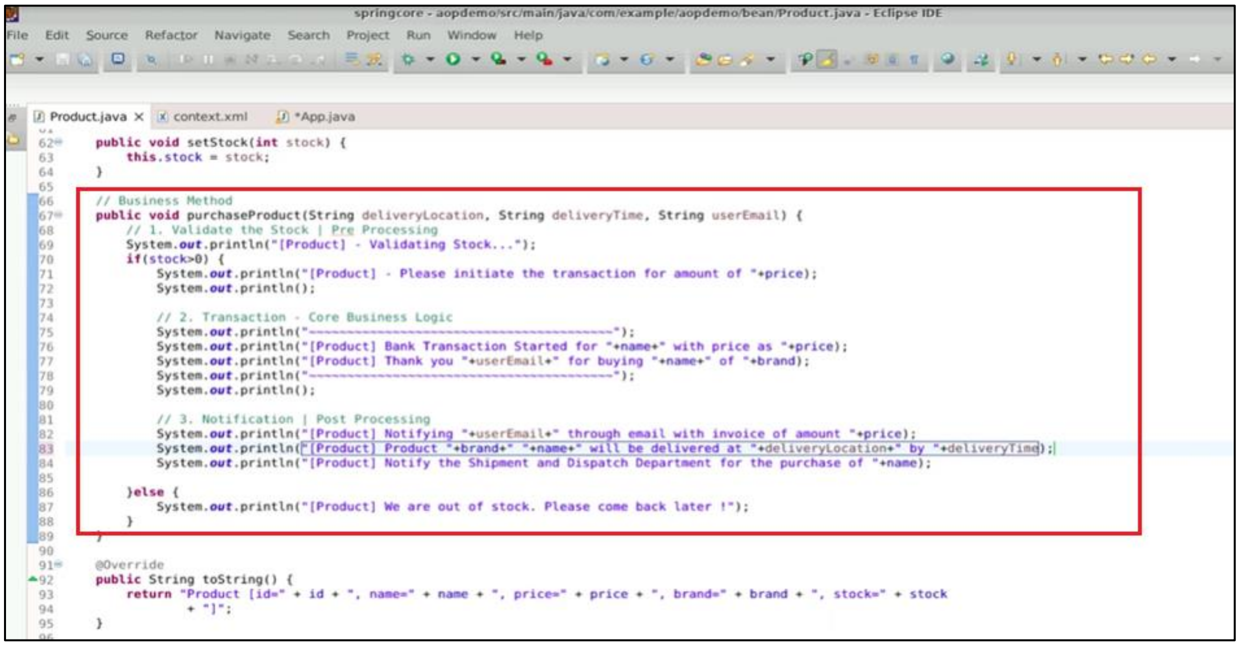
4.2 Now configure the properties for the **Product**, which are the key and value pair for each attribute of the **Product** class


springcore - aopdemo/src/main/java/context.xml - Eclipse IDE
File Edit Source Navigate Search Project Run Window Help
Product.java
context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <bean id="pRef" class="com.example.aopdemo.bean.Product">
        <property name="id" value="101"/>
        <property name="name" value="Water Bottle"/>
        <property name="price" value="2000"/>
        <property name="brand" value="Hidrate"/>
        <property name="stock" value="17"/>

    </bean>

</beans>
```

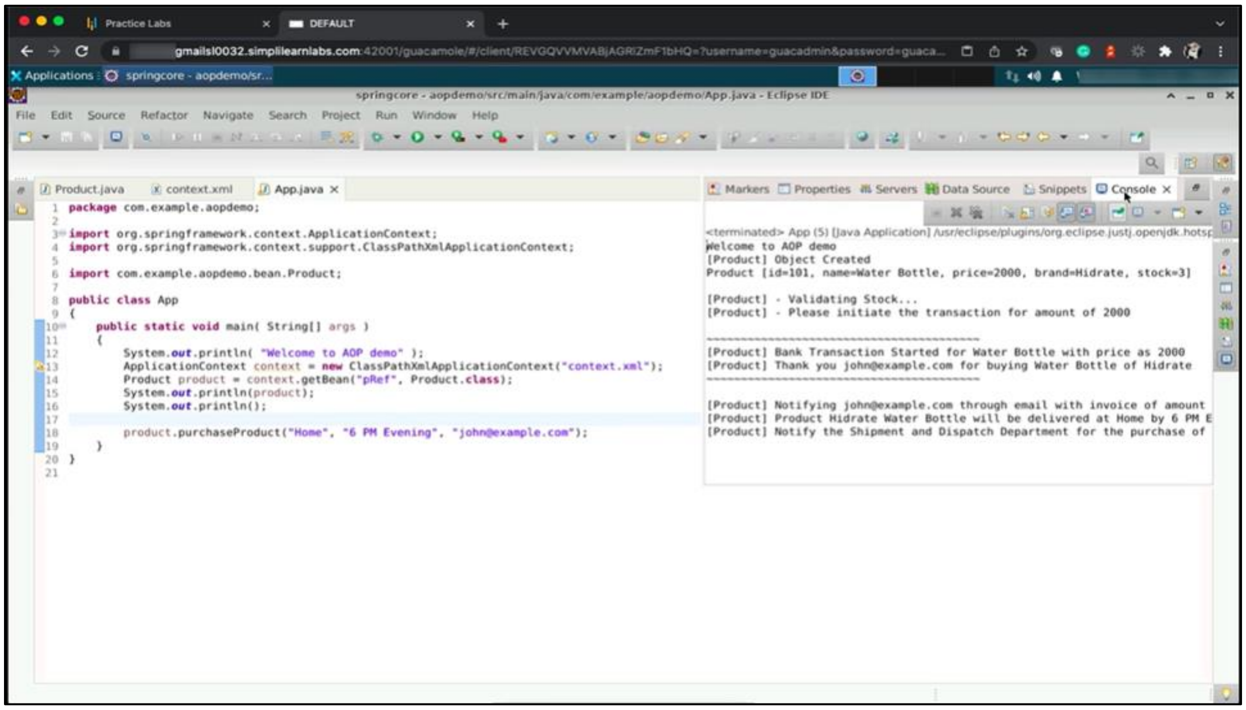
Step 5: Adding Business Methods

5.1 Add the following code to your **Product.java** file. This part will be your main business logic, which includes the pre-processing part, the actual business logic, and the post-processing part


springcore - aopdemo/src/main/java/com/example/aopdemo/bean/Product.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Product.java
context.xml
*App.java

```java
    public void setStock(int stock) {
        this.stock = stock;
    }

    // Business Method
    public void purchaseProduct(String deliveryLocation, String deliveryTime, String userEmail) {
        // 1. Validate the Stock | Pre Processing
        System.out.println("[Product] - Validating Stock...");
        if(stock>0) {
            System.out.println("[Product] - Please initiate the transaction for amount of "+price);
            System.out.println();

            // 2. Transaction - Core Business Logic
            System.out.println("----------------------------------------");
            System.out.println("[Product] Bank Transaction Started for "+name+" with price as "+price);
            System.out.println("[Product] Thank you "+userEmail+" for buying "+name+" of "+brand);
            System.out.println("----------------------------------------");
            System.out.println();

            // 3. Notification | Post Processing
            System.out.println("[Product] Notifying "+userEmail+" through email with invoice of amount "+price);
            System.out.println("[Product] Product "+brand+" "+name+" will be delivered at "+deliveryLocation+" by "+deliveryTime);
            System.out.println("[Product] Notify the Shipment and Dispatch Department for the purchase of "+name);

        }else {
            System.out.println("[Product] We are out of stock. Please come back later !");
        }
    }

    @Override
    public String toString() {
        return "Product [id=" + id + ", name=" + name + ", price=" + price + ", brand=" + brand + ", stock=" + stock
                + "]";
    }
```

5.2 Configure the **App.java** file to execute the code


springcore - aopdemo/src/main/java/com/example/aopdemo/App.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Product.java
context.xml
App.java
Markers Properties Servers Data Source Snippets Console

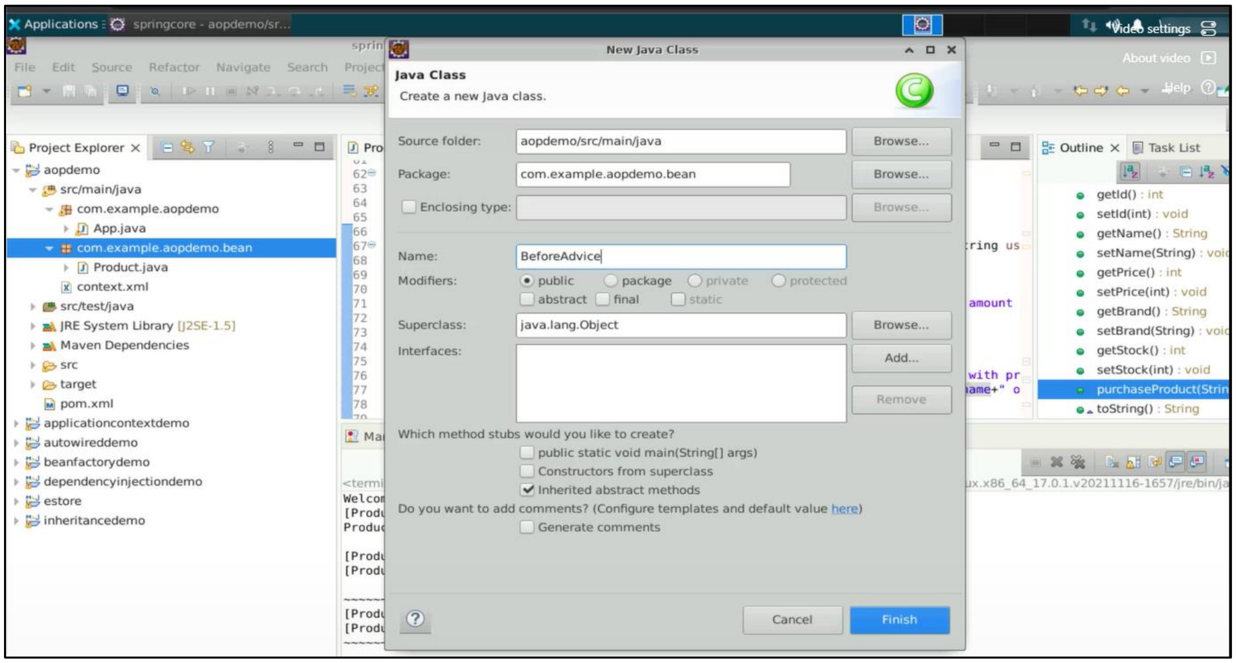
```java
package com.example.aopdemo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.aopdemo.bean.Product;

public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Welcome to AOP demo" );
        ApplicationContext context = new ClassPathXmlApplicationContext("context.xml");
        Product product = context.getBean("pRef", Product.class);
        System.out.println(product);
        System.out.println();

        product.purchaseProduct("Home", "6 PM Evening", "john@example.com");
    }
}
```

```
<terminated> App (5) [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.hotsp
Welcome to AOP demo
[Product] Object Created
Product [id=101, name=Water Bottle, price=2000, brand=Hidrate, stock=3]

[Product] - Validating Stock...
[Product] - Please initiate the transaction for amount of 2000

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[Product] Bank Transaction Started for Water Bottle with price as 2000
[Product] Thank you john@example.com for buying Water Bottle of Hidrate
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[Product] Notifying john@example.com through email with invoice of amount
[Product] Product Hidrate Water Bottle will be delivered at Home by 6 PM E
[Product] Notify the Shipment and Dispatch Department for the purchase of
```

Till now, the whole set of business logic methods has been kept under one class, which is the product class. But for bigger projects, it is more efficient to segregate different phases of business logic into different modules. So, let's create the before advice and the after returning advice.

Step 6: Creating Before and After Returning Advice

6.1 Go to the bean package and create a new class named **BeforeAdvice** to create the before advice method



New Java Class

Java Class

Create a new Java class.

Source folder: aopdemo/src/main/java

Package: com.example.aopdemo.bean

Enclosing type:

Name: BeforeAdvice

Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)

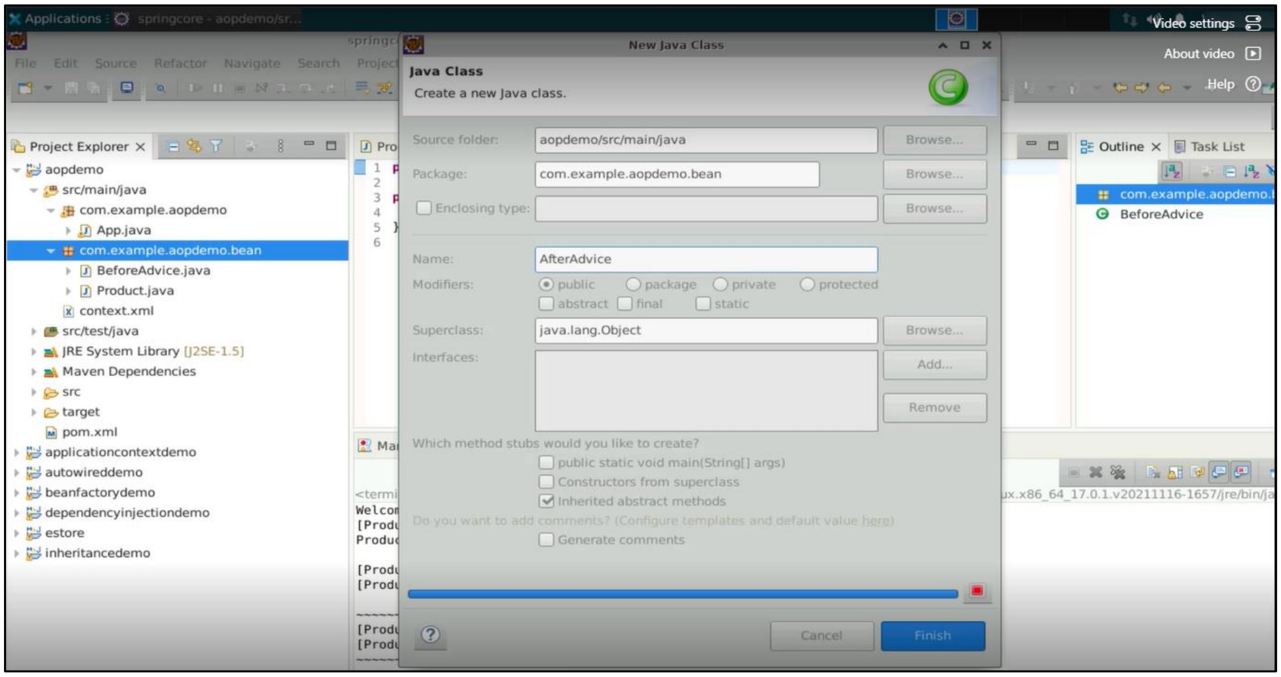
Constructors from superclass

[✓ Inherited abstract methods]

Do you want to add comments? (Configure templates and default value here)

Generate comments

6.2 Similarly, create another class **AfterAdvice** to create after returning advice



New Java Class

Java Class

Create a new Java class.

Source folder: aopdemo/src/main/java

Package: com.example.aopdemo.bean

Enclosing type:

Name: AfterAdvice

Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)

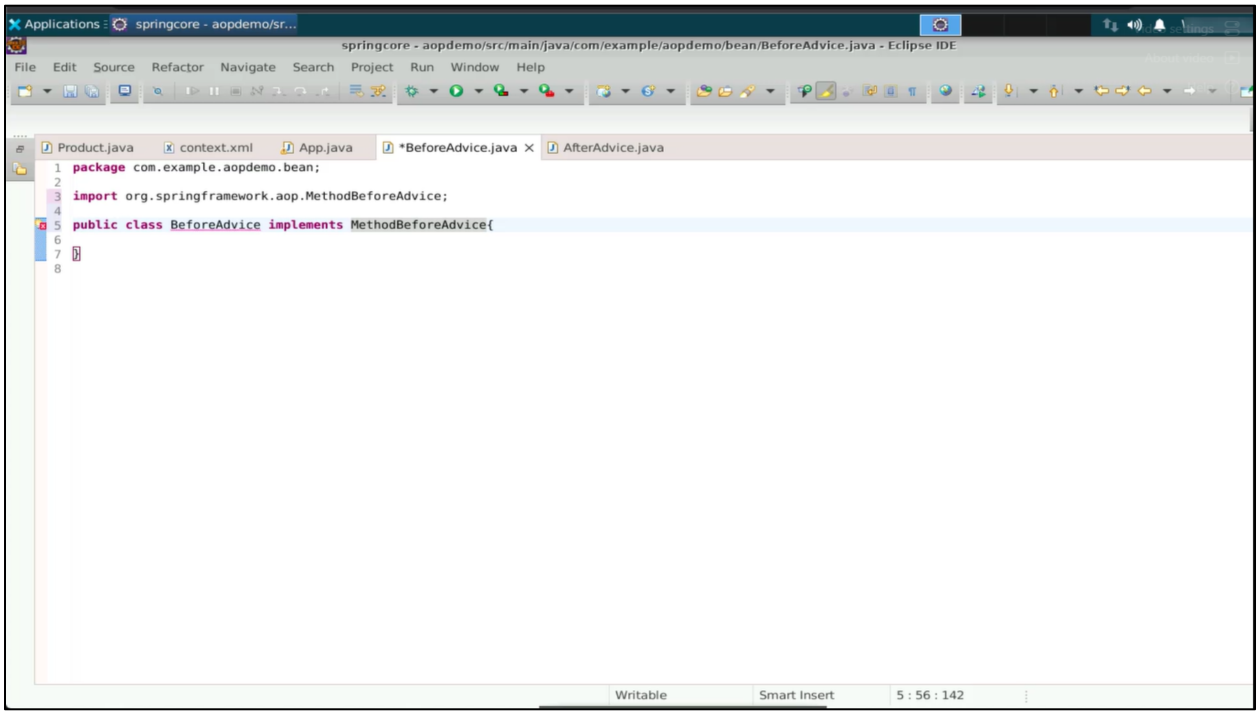
Constructors from superclass

[✓ Inherited abstract methods]

Do you want to add comments? (Configure templates and default value here)

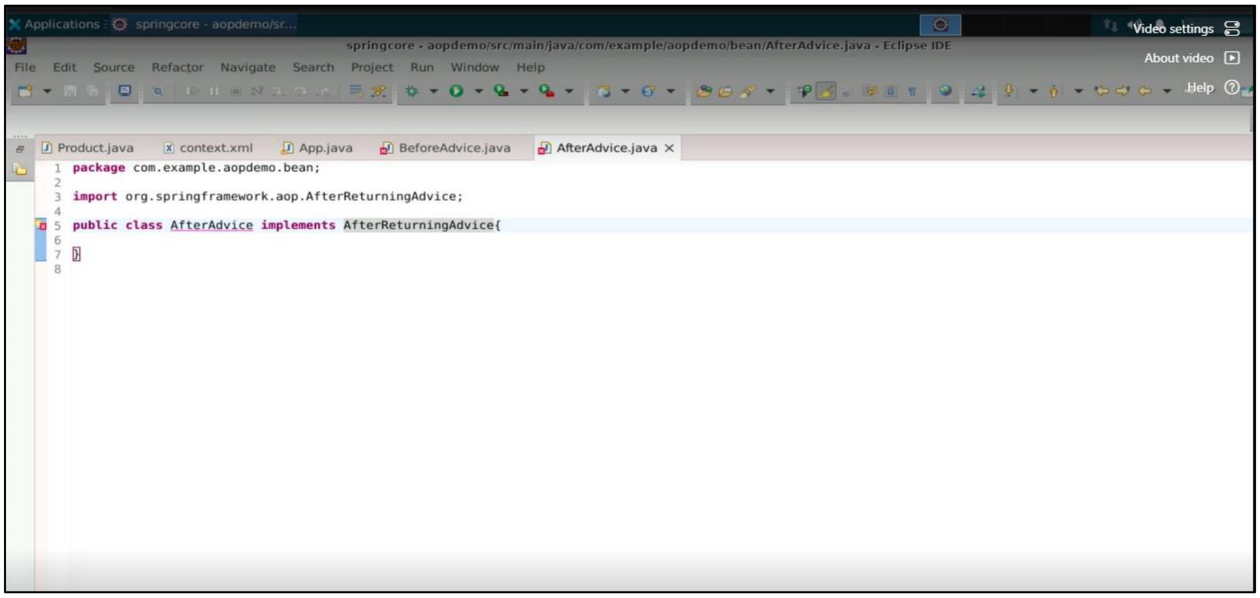
Generate comments

6.3 To implement **BeforeAdvice**, implement an interface **MethodBeforeAdvice** from Spring framework AOP Package


Applications
springcore - aopdemo/sr...
springcore - aopdemo/src/main/java/com/example/aopdemo/bean/BeforeAdvice.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Product.java
context.xml
App.java
*BeforeAdvice.java
AfterAdvice.java
Writable
Smart Insert
5 : 56 : 142

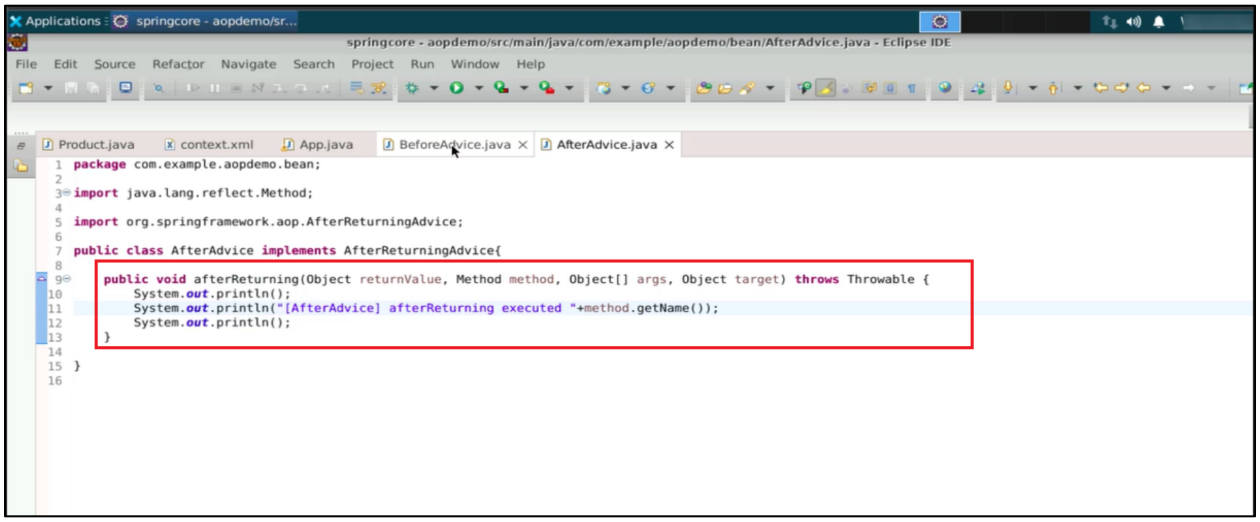
```
package com.example.aopdemo.bean;

import org.springframework.aop.MethodBeforeAdvice;

public class BeforeAdvice implements MethodBeforeAdvice{

}
```

6.4 To implement **AfterAdvice**, implement an interface **AfterReturningAdvice**


Applications
springcore - aopdemo/sr...
springcore - aopdemo/src/main/java/com/example/aopdemo/bean/AfterAdvice.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Video settings
About video
Help
Product.java
context.xml
App.java
BeforeAdvice.java
AfterAdvice.java

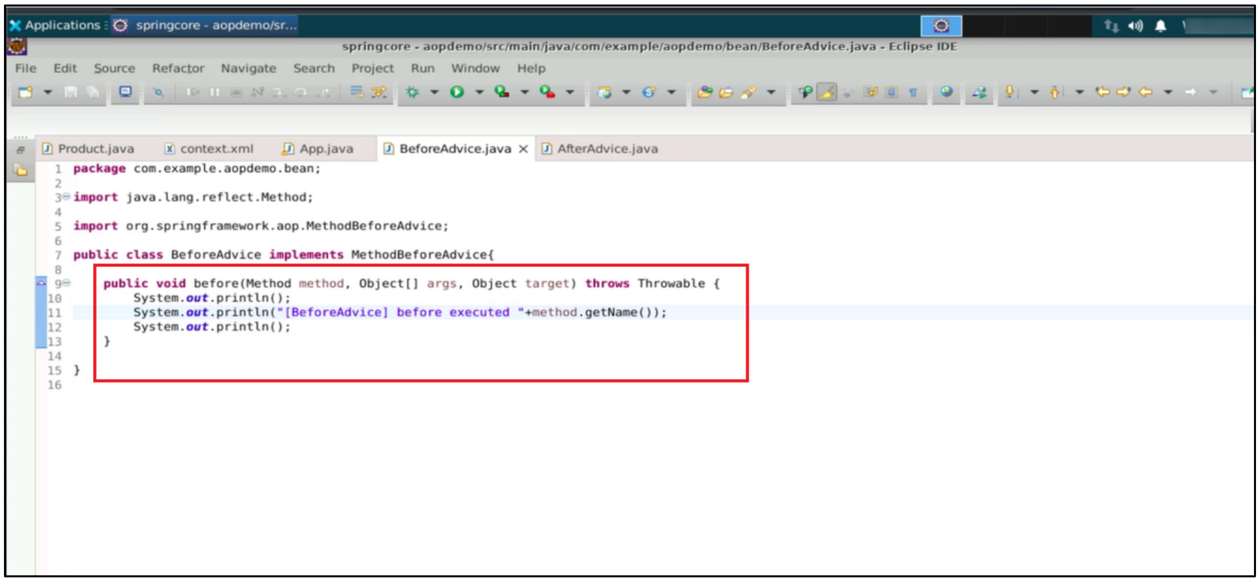
```
package com.example.aopdemo.bean;

import org.springframework.aop.AfterReturningAdvice;

public class AfterAdvice implements AfterReturningAdvice{

}
```

6.5 Add the highlighted code snippet under the **afterReturning** method to check when the after returning advice is executed while executing the whole project



```
package com.example.aopdemo.bean;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class AfterAdvice implements AfterReturningAdvice{

    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println();
        System.out.println("[AfterAdvice] afterReturning executed "+method.getName());
        System.out.println();
    }

}
```

6.6 Similarly, add the same under the **MethodBeforeAdvice** method

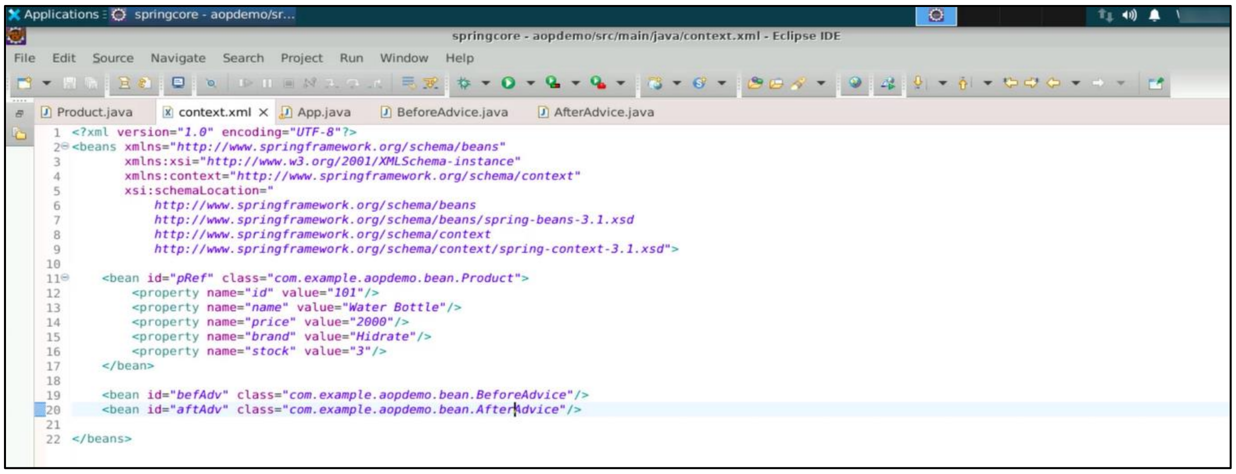


```
package com.example.aopdemo.bean;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class BeforeAdvice implements MethodBeforeAdvice{

    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println();
        System.out.println("[BeforeAdvice] before executed "+method.getName());
        System.out.println();
    }

}
```

With this, the before advice and after returning advice are successfully created within our project. Now, the next step is to configure this advice in the XML file.

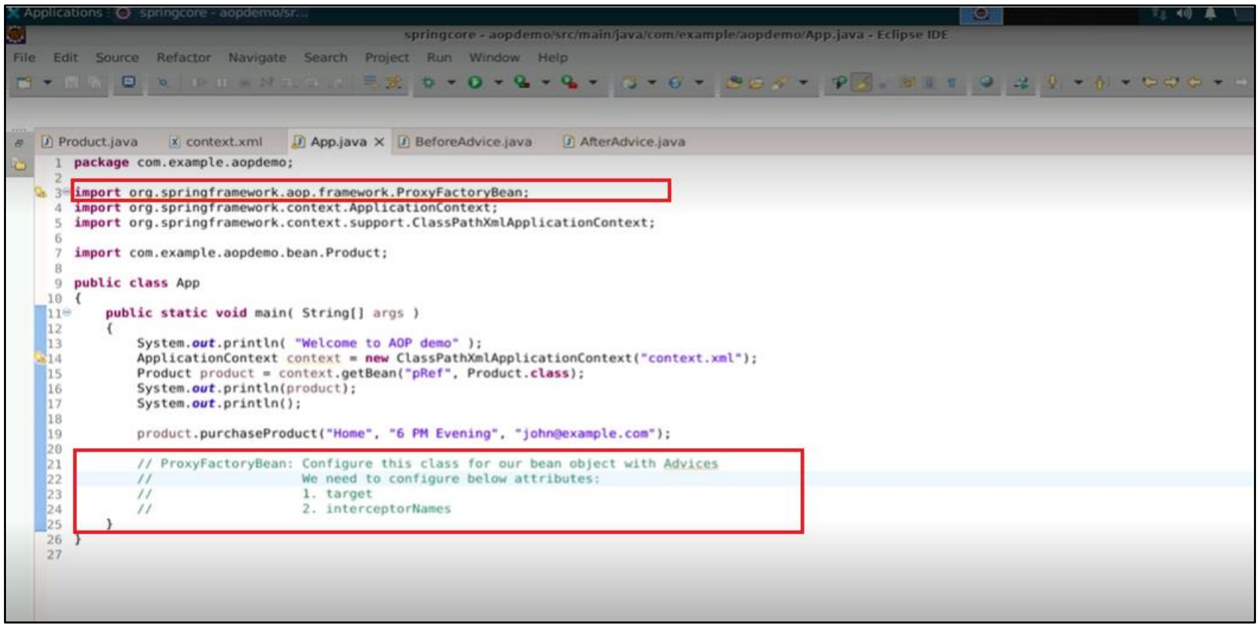
Step 7: Configuring both the advice in the XML file

7.1 Go to the **context.xml** file and add the two beans, which will create two objects by the IOC container


springcore - aopdemo/src/main/java/context.xml - Eclipse IDE
File Edit Source Navigate Search Project Run Window Help
Product.java
context.xml
App.java
BeforeAdvice.java
AfterAdvice.java

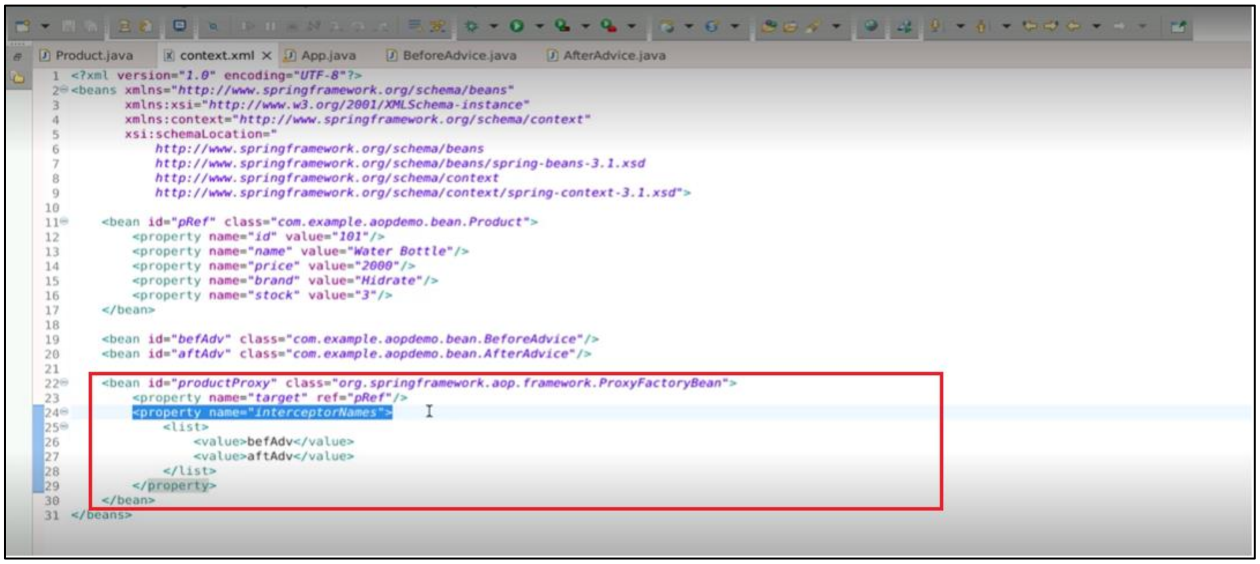
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <bean id="pRef" class="com.example.aopdemo.bean.Product">
        <property name="id" value="101"/>
        <property name="name" value="Water Bottle"/>
        <property name="price" value="2000"/>
        <property name="brand" value="Hidrate"/>
        <property name="stock" value="3"/>
    </bean>

    <bean id="befAdv" class="com.example.aopdemo.bean.BeforeAdvice"/>
    <bean id="aftAdv" class="com.example.aopdemo.bean.AfterAdvice"/>

</beans>
```

7.2 In the **App.java** file, import the Spring AOP framework to add the **ProxyFactoryBean** class and configure the attributes **target** and **interceptorNames** in the XML file


springcore - aopdemo/src/main/java/com/example/aopdemo/App.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Product.java
context.xml
App.java
BeforeAdvice.java
AfterAdvice.java

```
package com.example.aopdemo;

import org.springframework.aop.framework.ProxyFactoryBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.aopdemo.bean.Product;

public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Welcome to AOP demo" );
        ApplicationContext context = new ClassPathXmlApplicationContext("context.xml");
        Product product = context.getBean("pRef", Product.class);
        System.out.println(product);
        System.out.println();

        product.purchaseProduct("Home", "6 PM Evening", "john@example.com");

        // ProxyFactoryBean: Configure this class for our bean object with Advices
        //                   We need to configure below attributes:
        //                   1. target
        //                   2. interceptorNames
    }
}
```

7.3 Create another bean of class **org.springframework.aop.framework.ProxyFactoryBean** in the **context.xml** file, then add all the necessary configurations for the attributes of the **ProxyFactoryBean** class

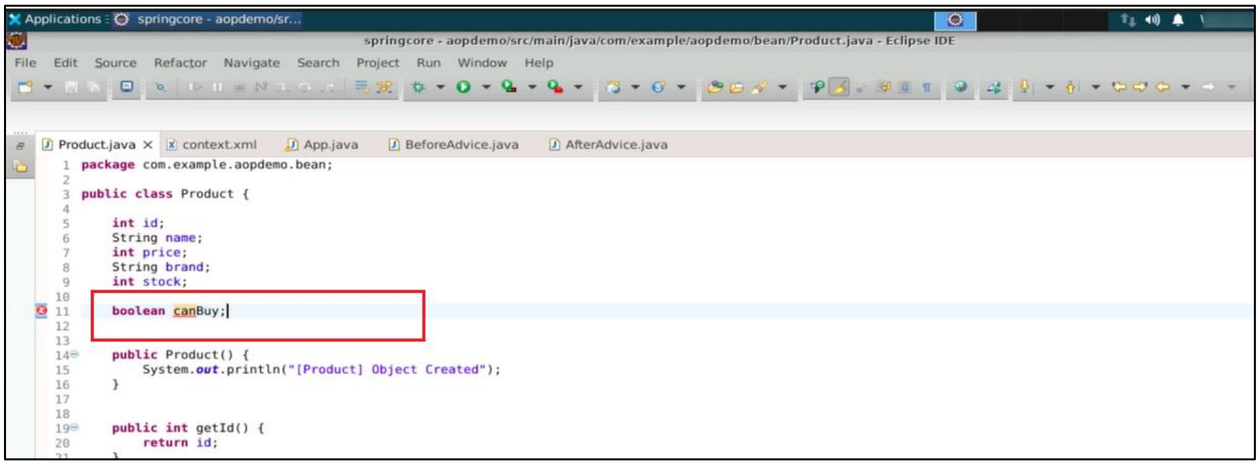


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.1.xsd">

    <bean id="pRef" class="com.example.aopdemo.bean.Product">
        <property name="id" value="101"/>
        <property name="name" value="Water Bottle"/>
        <property name="price" value="2000"/>
        <property name="brand" value="Hidrate"/>
        <property name="stock" value="3"/>
    </bean>

    <bean id="befAdv" class="com.example.aopdemo.bean.BeforeAdvice"/>
    <bean id="aftAdv" class="com.example.aopdemo.bean.AfterAdvice"/>

    <bean id="productProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="pRef"/>
        <property name="interceptorNames">
            <list>
                <value>befAdv</value>
                <value>aftAdv</value>
            </list>
        </property>
    </bean>
</beans>
```

Finally, the last step is to divide different phases of business logic into their respective advice in Spring AOP.

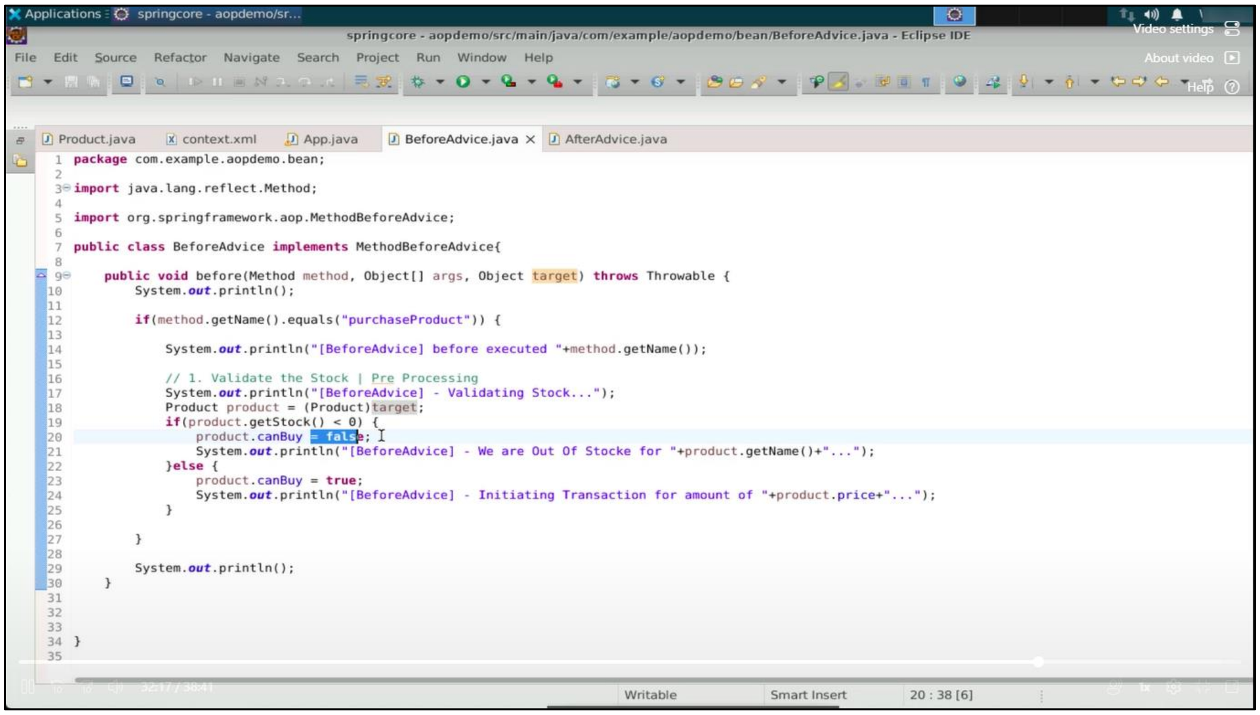
Step 8: Dividing the business logic into before and after returning advice

8.1 First, comment on the whole business method section and add a Boolean variable **canBuy** which will act as a validating attribute



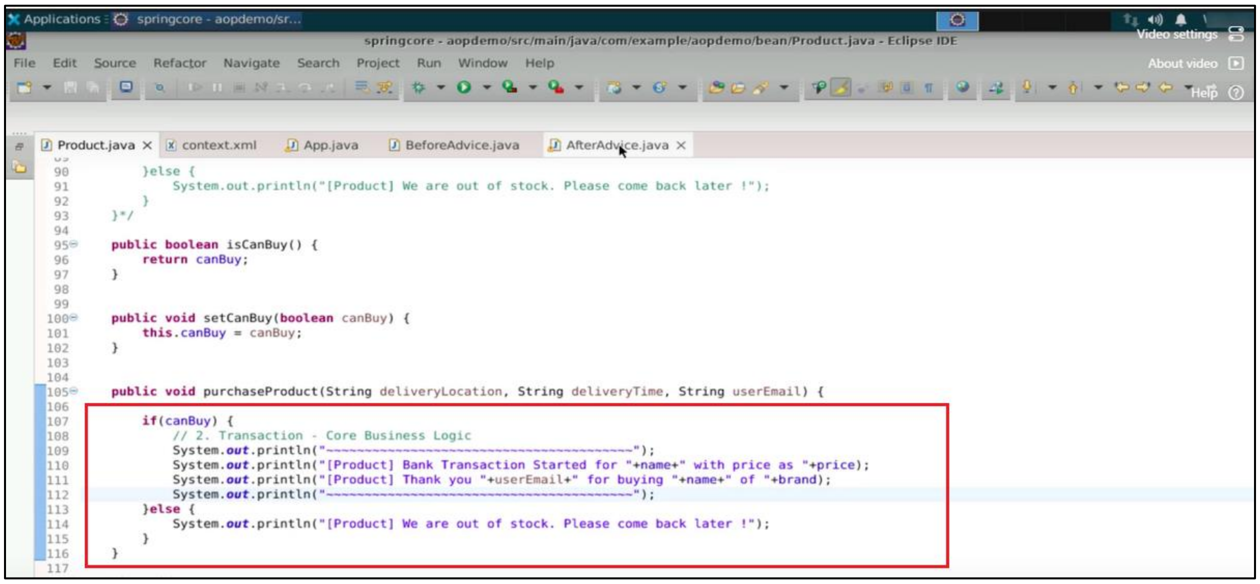
```
package com.example.aopdemo.bean;

public class Product {

    int id;
    String name;
    int price;
    String brand;
    int stock;

    boolean canBuy;


    public Product() {
        System.out.println("[Product] Object Created");
    }


    public int getId() {
        return id;
```

8.2 Generate the **Getters and Setters** for **canBuy** and add the preprocessing part of the business logic in the **BeforeAdvice.java**



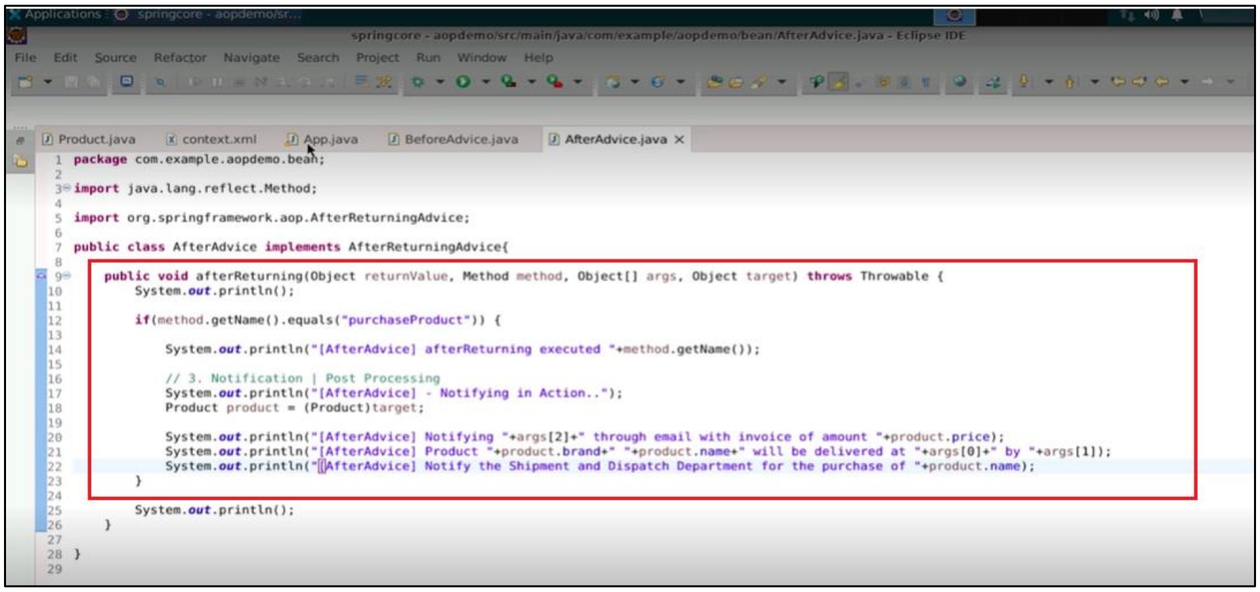
```java
package com.example.aopdemo.bean;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class BeforeAdvice implements MethodBeforeAdvice{

    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println();

        if(method.getName().equals("purchaseProduct")) {

            System.out.println("[BeforeAdvice] before executed "+method.getName());

            // 1. Validate the Stock | Pre Processing
            System.out.println("[BeforeAdvice] - Validating Stock...");
            Product product = (Product)target;
            if(product.getStock() < 0) {
                product.canBuy = false;
                System.out.println("[BeforeAdvice] - We are Out Of Stocke for "+product.getName()+"...");
            }else {
                product.canBuy = true;
                System.out.println("[BeforeAdvice] - Initiating Transaction for amount of "+product.price+"...");
            }

        }

        System.out.println();
    }



}
```

8.3 Now remove the preprocessing and post-processing parts from **Product.java**. Add the following logic as the core business logic part.



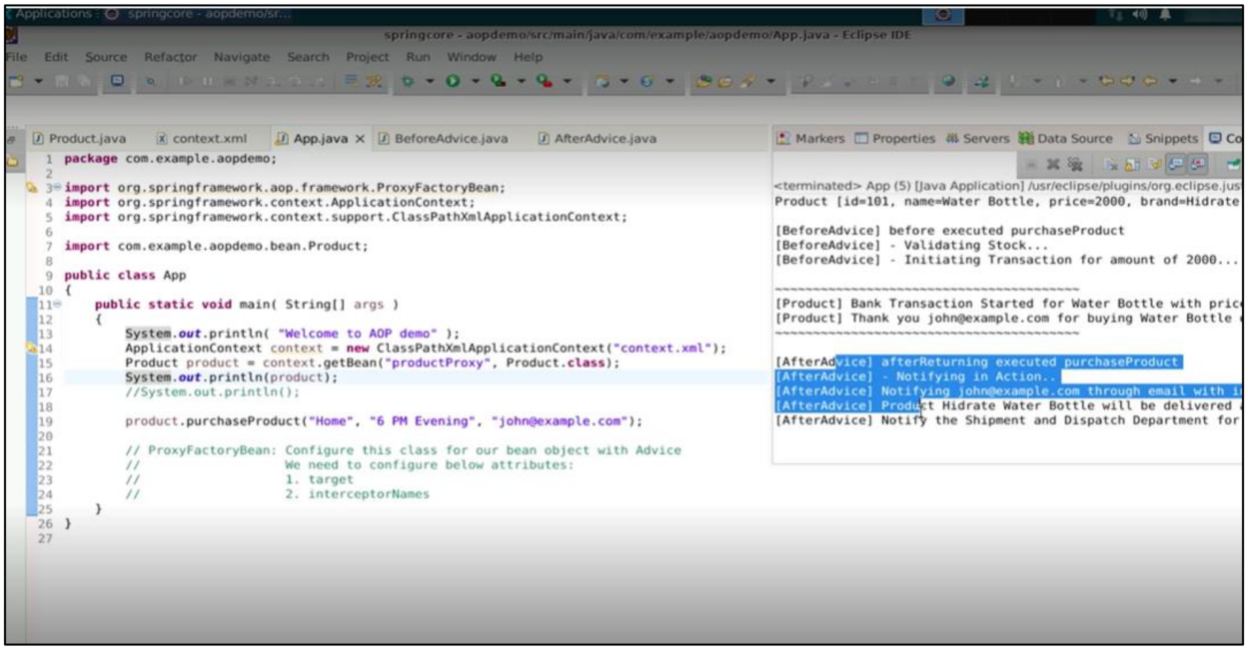
```java
        }else {
            System.out.println("[Product] We are out of stock. Please come back later !");
        }
    }*/

    public boolean isCanBuy() {
        return canBuy;
    }


    public void setCanBuy(boolean canBuy) {
        this.canBuy = canBuy;
    }


    public void purchaseProduct(String deliveryLocation, String deliveryTime, String userEmail) {

        if(canBuy) {
            // 2. Transaction - Core Business Logic
            System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
            System.out.println("[Product] Bank Transaction Started for "+name+" with price as "+price);
            System.out.println("[Product] Thank you "+userEmail+" for buying "+name+" of "+brand);
            System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        }else {
            System.out.println("[Product] We are out of stock. Please come back later !");
        }
    }

```

8.4 Add the post-processing part of the business logic in the **AfterAdvice.java**



```java
package com.example.aopdemo.bean;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class AfterAdvice implements AfterReturningAdvice{

    public void afterReturning(Object returnValue, Method method, Object[] args, Object target) throws Throwable {
        System.out.println();

        if(method.getName().equals("purchaseProduct")) {

            System.out.println("[AfterAdvice] afterReturning executed "+method.getName());

            // 3. Notification | Post Processing
            System.out.println("[AfterAdvice] - Notifying in Action..");
            Product product = (Product)target;

            System.out.println("[AfterAdvice] Notifying "+args[2]+" through email with invoice of amount "+product.price);
            System.out.println("[AfterAdvice] Product "+product.brand+" "+product.name+" will be delivered at "+args[0]+" by "+args[1]);
            System.out.println("[AfterAdvice] Notify the Shipment and Dispatch Department for the purchase of "+product.name);
        }

        System.out.println();
    }

}
```

8.5 Lastly, execute the project by making the necessary changes in **App.java**



```java
package com.example.aopdemo;

import org.springframework.aop.framework.ProxyFactoryBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.aopdemo.bean.Product;

public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Welcome to AOP demo" );
        ApplicationContext context = new ClassPathXmlApplicationContext("context.xml");
        Product product = context.getBean("productProxy", Product.class);
        System.out.println(product);
        //System.out.println();

        product.purchaseProduct("Home", "6 PM Evening", "john@example.com");

        // ProxyFactoryBean: Configure this class for our bean object with Advice
        //                   We need to configure below attributes:
        //                   1. target
        //                   2. interceptorNames
    }
}
```

```
<terminated> App (5) [Java Application] /usr/eclipse/plugins/org.eclipse.jus
Product [id=101, name=Water Bottle, price=2000, brand=Hidrate

[BeforeAdvice] before executed purchaseProduct
[BeforeAdvice] - Validating Stock...
[BeforeAdvice] - Initiating Transaction for amount of 2000...

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[Product] Bank Transaction Started for Water Bottle with pric
[Product] Thank you john@example.com for buying Water Bottle
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[AfterAdvice] afterReturning executed purchaseProduct
[AfterAdvice] - Notifying in Action..
[AfterAdvice] Notifying john@example.com through email with i
[AfterAdvice] Product Hidrate Water Bottle will be delivered
[AfterAdvice] Notify the Shipment and Dispatch Department for
```

With this, the modularization of a business problem has been successfully done by implementing Aspect-Oriented Programming in the Spring Framework using Before and After Returning Advice.