Spring - AOP Example (Spring1.2 Old Style AOP)

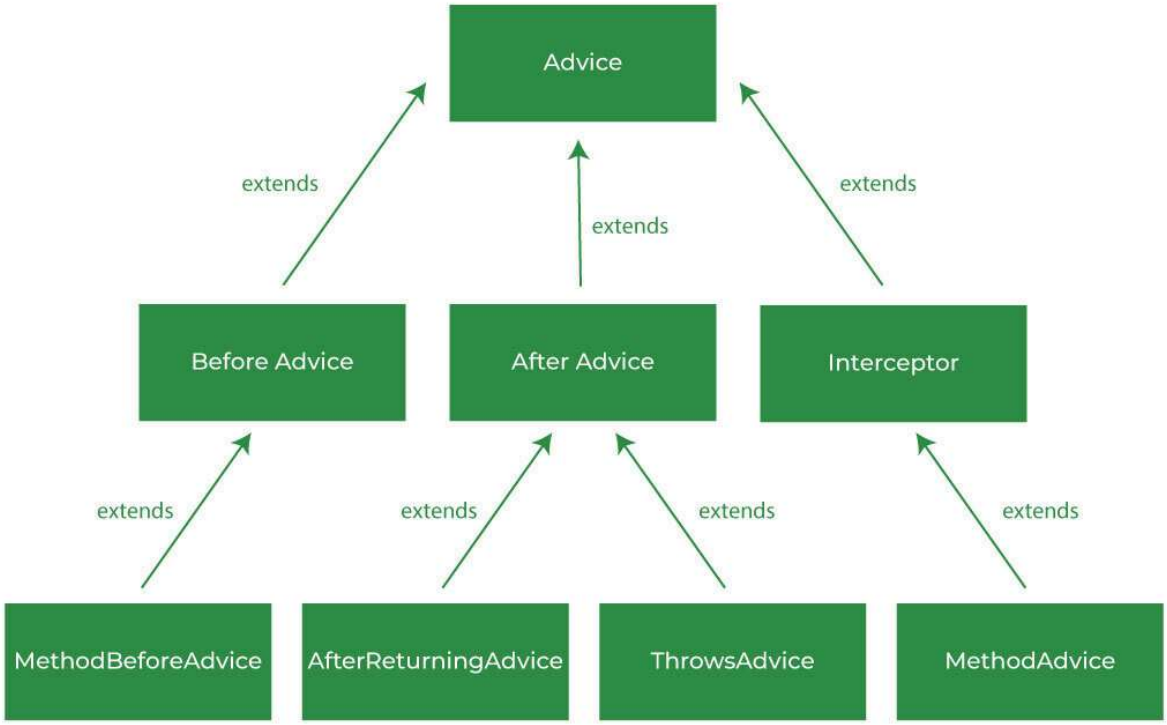
Last Updated : 23 Jul, 2025

Spring AOP allows adding additional behavior to the existing code without its modification. The Spring 1.2 Old Style AOP is supported in Spring 3 but there are a few things such as advice domain objects, that can not be done easily with Spring AOP. The AspectJ is the best available option in that case. And so, it is recommended to use Spring AOP with AspectJ.

The most important module of Aspect-Oriented Programming is advice. Advice is an action performed by an aspect at a particular join-points. There are 4 types of advices in Spring 1.2 Old Style AOP:

1. **Before Advice:** It is executed before the actual method call.
2. **After Advice:** It is executed after the actual method call.
3. **Around Advice:** It is executed before and after the actual method call.
4. **Throws Advice:** It is executed if the actual method throws an exception.

The architecture of Spring AOP Advice is as follows:



Architecture Of Spring AOP

A. MethodBeforeAdvice

It is shown step by step that is as follows:

Step 1: Create a business logic class

In this step, we will create a class that will contain actual business logic. For this, we will create a new class and name it Geeks. And add the following line of codes to it.

```
// Class
public class Geeks {

    // for actual business logic class we will
    // define a public method to print some random text
    public void message() {
        System.out.println("Welcome to GeeksforGeeks!!");
    }
}
```

Step 2: Creating Advisor class

In this step, we will create an Advisor class and name it as **BeforeAdvisor,** and do remember this class will implement the interface **MethodBeforeAdvice**.

Example:

```
// Java Program to Illustrate BeforeAdvisor Class

// Importing required classes
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

// Class
public class BeforeAdvisor implements MethodBeforeAdvice {

    // Method
    // Executes before actual business logic call
    public void before(Method arg0, Object[] arg1,
                       Object arg2) throws Throwable
    {

        // Print statement
        System.out.println(
            "This is called before the actual call..");
    }
}
```

Before moving further we need to know about **ProxyFactoryBean**. It is a class provided by the Spring framework. ProxyFactoryBean class contains 2 fields target and interceptorName. For our example, the instance of Geeks class will be considered as target and the instance of advisor class as an interceptor.

Step 3: Create an application-context.xml file

In this step, we will create beans in the context file for our Geeks, Advisor, and ProxyFactoryBean class.

> ***Note:*** *You need to provide your class path and the package definition while creating beans.*

File: application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans-3.0.xsd">

    <bean id="geeksObj" class="com.geeksforgeeks.Geeks"></bean>
    <bean id="beforeAdvisorObj" class="com.geeksforgeeks.BeforeAdvisor">
</bean>

    <bean id="proxyObj"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="geeksObj"></property>
        <property name="interceptorNames">
            <list>
            <value>beforeAdvisorObj</value>
            </list>
        </property>
    </bean>

</beans>
```

Step 4: Adding dependencies

Here, we will add the required dependencies in our **pom.xml** file.

File: pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.geeksforgeeks</groupId>
  <artifactId>SpringAOP</artifactId>
  <version>0.0.1-SNAPSHOT</version>

<dependencies>

    <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.3.RELEASE</version>
    </dependency>

    <!-- <a
```

```
href="https://mvnrepository.com/artifact/aspectj/aspectjrt">https://mvnreposi
tory.com/artifact/aspectj/aspectjrt</a> -->
    <dependency>
    <groupId>aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.5.4</version>
    </dependency>

    <!-- <a
href="https://mvnrepository.com/artifact/org.aspectj/aspectjweaver">https://m
vnrepository.com/artifact/org.aspectj/aspectjweaver</a> -->
    <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.2</version>
    </dependency>

    <!-- <a
href="https://mvnrepository.com/artifact/org.aspectj/aspectjtools">https://mv
nrepository.com/artifact/org.aspectj/aspectjtools</a> -->
    <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjtools</artifactId>
    <version>1.9.2</version>
    </dependency>

    <!-- <a
href="https://mvnrepository.com/artifact/org.ow2.asm/asm">https://mvnreposito
ry.com/artifact/org.ow2.asm/asm</a> -->
    <dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm</artifactId>
    <version>7.0</version>
    </dependency>

    <!-- <a
href="https://mvnrepository.com/artifact/org.springframework/spring-
aspects">https://mvnrepository.com/artifact/org.springframework/spring-
aspects</a> -->
    <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.1.3.RELEASE</version>
    </dependency>

  </dependencies>
</project>
```

Step 5: Creating application/main file

In this step, we will create a Test.java class and initialize the beans and calls message() method of Geeks.java class.

File: Test.java (Main File/Application File)

```
// Java Program to Illustrate Application Class

// Importing required classes
import java.util.ResourceBundle;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

// Main class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {
        // Reading the application-context file using
        // class path of spring context xml file
        Resource context = new ClassPathResource(
            "application-context.xml");

        BeanFactory beanFactory
            = new XmlBeanFactory(context);

        // Initializing geeks bean using bean factory
        Geeks geeks
            = beanFactory.getBean("proxyObj", Geeks.class);

        // Calling the message() method
        // inside main() method
        geeks.message();
    }
}
```

Step 6: Appending output via running application file.

In this step, we will run our application and get the output.

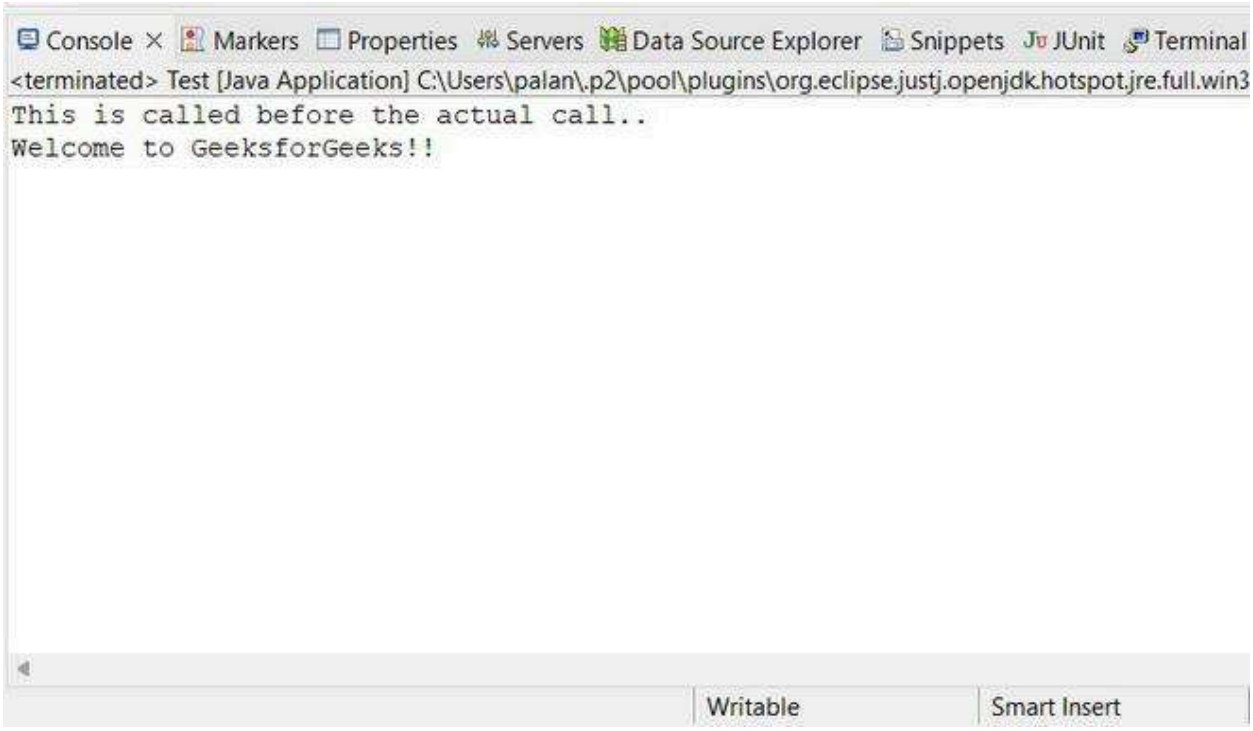


Fig 2 - Output

B. AfterReturningAdvice

Implementation:

Step 1: Create a business logic class

We will be using the same Geeks.java class for actual business logic.

Step 2: Create AfterAdvisor class

In this step, we will create a class and name it AfterAdvisor.java and implement the **AfterReturningAdvice** interface and implements its method **afterReturning()**.

```
// Java Program to Illustrate AfterAdvisor Class

// Importing required classes
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

// Class
public class AfterAdvisor
implements AfterReturningAdvice
```

```
{
    // Method
    // Executes after the actual business logic call
    public void afterReturning(Object returnValue, Method method,
Object[] args, Object target)
    throws Throwable
    {
        System.out.println("This is called after returning
advice..");
    }
}
```

Step 3: Changing the application-context.xml file

Here we will be updating our application-context.xml file according to the **AfterAdvisor** class and **AfterReturningAdvice** interface.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans-3.0.xsd">

    <bean id="geeksObj" class="com.geeksforgeeks.Geeks"></bean>
    <bean id="afterAdvisorObj" class="com.geeksforgeeks.AfterAdvisor">
</bean>

    <bean id="proxyObj"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="geeksObj"></property>
        <property name="interceptorNames">
            <list>
            <value>afterAdvisorObj</value>
            </list>
        </property>
    </bean>

</beans>
```

Step 4: Output

In this step, we will run our application.

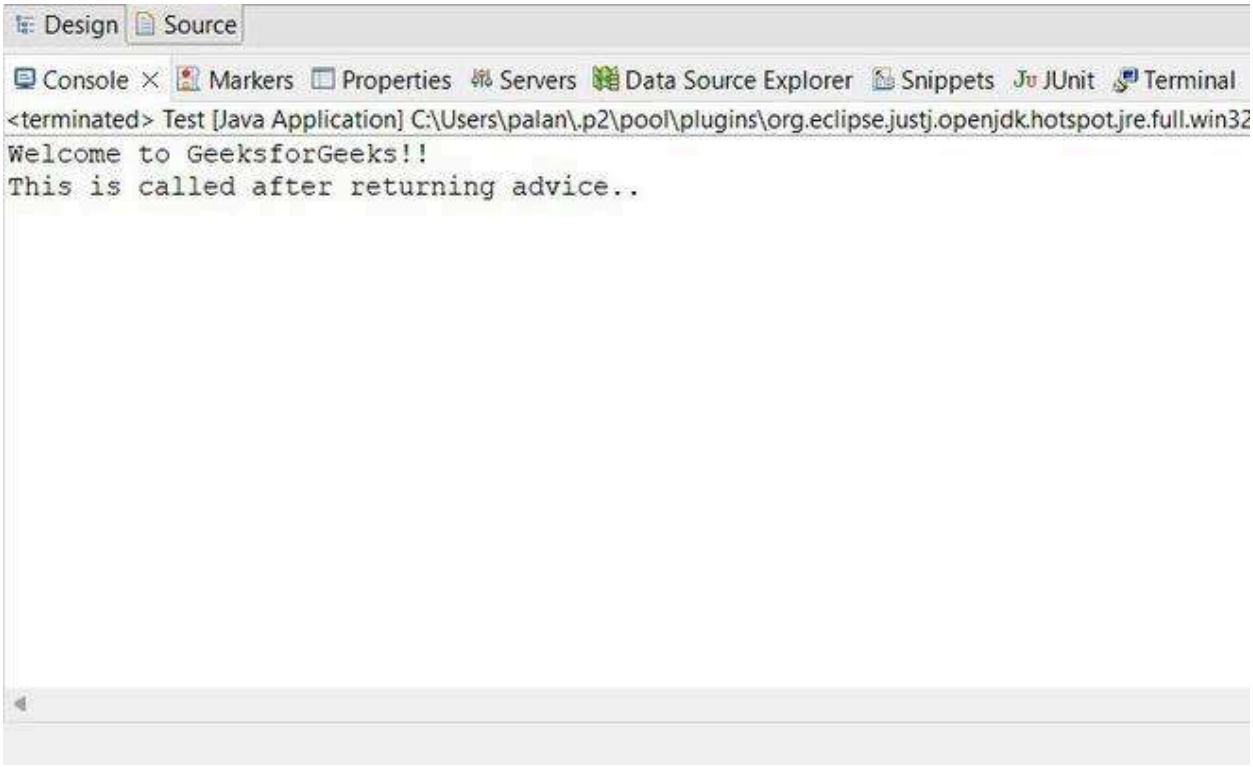


Fig 3 - Output

C. ThrowsAdvice

Procedure: It is as Implementation below step by step as shown below as follows:

Step 1: Create a business logic class

Here we will be updating our business logic class Geeks.java. Below is the code for Geeks.java class.

```
// Class
public class Geeks {

    // For actual business logic class we will
    // define a public method to print some random text
    public void message(String str) {

        // Checking if string is equal or not
        if (str.equals("geeksforgeeks.org"))

            // Print statement as string is equal
            System.out.println("Welcome to GeeksforGeeks!!");
```

```
        else

            // Do throw exception
            throw new IllegalArgumentException("Invalid String");
    }
}
```

Step 2: Create ThrowsAdvisor class

In this step, we will create a class and name it 'ThrowsAdvisor.java' and we will implement the **ThrowsAdvice** interface and implement the **afterThrowing()** method. Here, we will put those concerns which will need to be thrown even if the exception has occurred.

Example

```
// Java Program to Illustrate ThrowsAdvice Class

// Importing required classes
import org.springframework.aop.ThrowsAdvice;

// Class
public class ThrowsAdvisor implements ThrowsAdvice {

    // Method
    public void afterThrowing(Exception e)
    {
        // Print statement
        System.out.println(
            "This will be called if exception occurs..");
    }
}
```

Step 3: Changing the application-context.xml file

In this step, we will update our application-context.xml file according to ThrowsAdvisor.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans-3.0.xsd">

    <bean id="geeksObj" class="com.geeksforgeeks.Geeks"></bean>
```

```
    <bean id="throwsAdvisorObj" class="com.geeksforgeeks.ThrowsAdvisor">
</bean>

    <bean id="proxyObj"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="geeksObj"></property>
        <property name="interceptorNames">
            <list>
            <value>throwsAdvisorObj</value>
            </list>
        </property>
    </bean>

</beans>
```

Step 4: Creating Test.java class

In this step, we will create a new class and name 'Test.java'. In this class, we will initialize our beans and call ***message(String str) method*** of our business logic class Geeks.java. We will pass an invalid string so, our application will throw an exception and still call ThrowsAdvice.

File: Test.java (Application Class)

```
// Java Program to Illustrate Application Class

// Importing required classes
import java.util.ResourceBundle;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

// Main class
// Application Class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {
        // Reading the application-context file
        // using class path of spring context xml file
        Resource context = new ClassPathResource(
            "application-context.xml");

        BeanFactory beanFactory
            = new XmlBeanFactory(context);

        // Initializing geeks bean
        // using bean factory
```

```
        Geeks geeks
            = beanFactory.getBean("proxyObj", Geeks.class);

        // Calling the message() method
        // inside main() method

        // Try block to check for exceptions
        try {
            geeks.message("gfg");
        }

        // Catch block to handle exceptions
        catch (Exception e) {

            // Display exceptions using line number
            // using printStackTrace() method
            e.printStackTrace();
        }
    }
}
```

Step 5: Run the application.

Output:

Fig 4 - Output

D. MethodInterceptor(AroundAdvice)

Procedure:

It is as Implementation below step by step as shown below as follows:

Step 1: Creating a business logic class

In this step, we will update our business logic class Geeks.java. Below is the code for Geeks.java class.

Example

```
// Class
public class Geeks {

    // Method
    // Define a public method to print some random text
    // for actual business logic class we will
    public void message()
    {

        // Print statement
        System.out.println("This is around advice..");
    }
}
```

Step 2: Create AroundAdvice class

In this step, we will create a new class and name as AroundAdvice and with this class, we will implement the ***'MethodInterceptor interface***' and provide the definition for invoke() method.

Example

```
// Java Program to Illustrate AroundAdvice Class

// Importing required classes
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

// Class
public class AroundAdvice implements MethodInterceptor {

    // Method
    public Object invoke(MethodInvocation invocation)
        throws Throwable
    {
        Object obj;

        System.out.println(
            "This will be run before actual business logic...");
        obj = invocation.proceed();
```

```
        // Print statement
        System.out.println(
            "This will be run after actual business logic...");

        return obj;
    }
}
```

Step 3: Update the application-context.xml file

Now, we will change our application-context.xml file according to AroundAdvice.

File: application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans/"
    xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans/
    http://www.springframework.org/schema/beans//spring-beans-3.0.xsd">

    <bean id="geeksObj" class="com.geeksforgeeks.Geeks"></bean>
    <bean id="aroundAdvisorObj" class="com.geeksforgeeks.AroundAdvice">
</bean>

    <bean id="proxyObj"
class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="geeksObj"></property>
        <property name="interceptorNames">
            <list>
            <value>aroundAdvisorObj</value>
            </list>
        </property>
    </bean>
</beans>
```

Step 4: Create Test.java file

In this step, we will use our previous ***'Test.java'*** and update it according to the business requirement.

FIle: Test.java

```
// Java Program to Illustrate Test Class

// Importing required classes
import java.util.ResourceBundle;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

// Class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {
        // Reading the application-context file using
        // class path of spring context xml file
        Resource context = new ClassPathResource(
            "application-context.xml");

        BeanFactory beanFactory
            = new XmlBeanFactory(context);

        // Initializing geeks bean using bean factory
        Geeks geeks
            = beanFactory.getBean("proxyObj", Geeks.class);

        // Calling message() method
        // inside() main() method
        geeks.message();
    }
}
```

Step 5: Output

In this step, we will run our application.

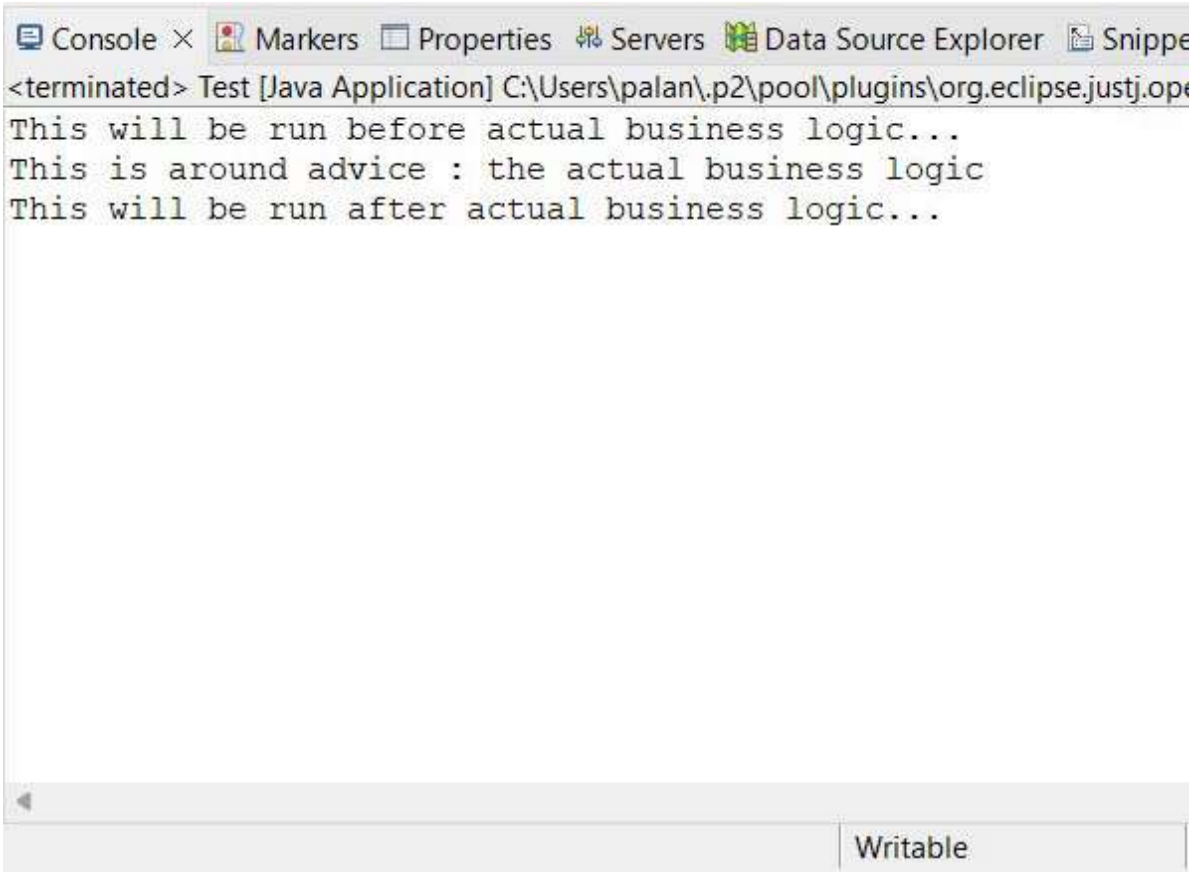


Fig 5 - Output

Comment More info

Campus Training Program

GeeksforGeeks
Sanchhaya Education Private Limited

Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:

K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305