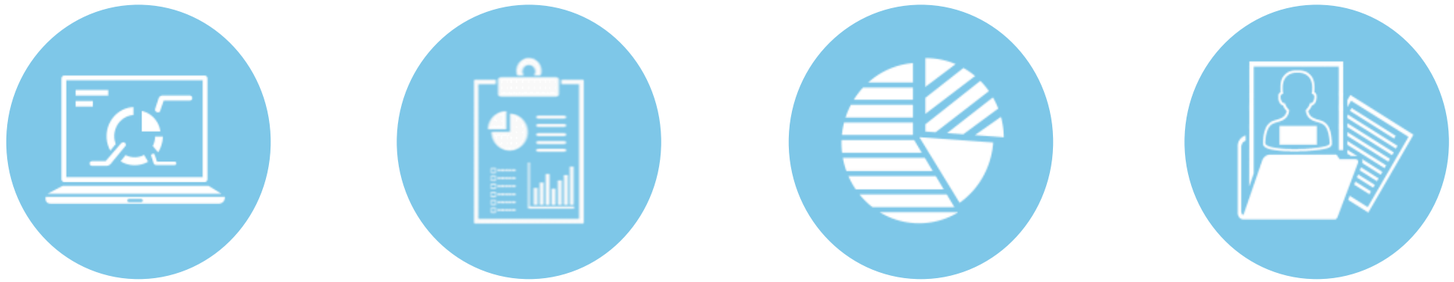
Advanced Java

Lesson 8—Spring AOP



Learning Objectives

- Explain AOP and its terminology
- Configure AOP in a Java Application using AspectJ Approach

Advanced Java

Topic 1—Introduction to Spring AOP (Aspect-Oriented Programming)

Why AOP?

AOP is used in applications that have cross-cutting **concerns,** that is, pieces of logic or code that are written in multiple classes/layers as per the requirements.

Common examples:

- Transaction Management
- Logging
- Exception Handling (especially when you may want to have detailed traces or have some plan of recovering from exceptions)
- Security aspects
- Instrumentation

Understanding Concerns

A concern is a piece of code that performs a specific task. There are two types of concerns:

1) “Core” concerns are codes used for business logic

2) “Cross concerns” are functions that are conceptually separate from the application's business logic but affect the entire service layer.

   Example: logging, auditing, declarative transactions, security, caching

Aspect-Oriented Programming provides one way to decouple dynamically core concern from crosscutting concern.

What Is AOP?

1) It is one of the basic components of Spring framework.

2) The main idea of AOP (Aspect-Oriented Programing) is to isolate the cross-cutting concerns from the application code, thereby modularizing them as a different entity.

3) Services layer deals with two types of concerns: Core and cross cutting concerns.

Uses of AOP

- Spring AOP provides declarative enterprise services such as declarative transaction management.
- It allow users to implement custom aspects.
- Spring AOP is used to track user activity in large business applications.
- It uses Proxy as a mechanism to implement cross-cutting concerns in a non-intrusive way.

AOP Terminologies

- Aspect: A feature or functionality that cross-cuts over objects. It has a set of APIs (Application Programming Interface) that provides cross-cutting requirements.
- JoinPoint: It defines the various execution points where an Aspect can be applied
- Pointcut: A Pointcut tells us about the Join Points where Aspects will be applied
- Advice: An advice provides concrete code implementation for the Aspect
- Target: It is a business object that comprises of Aspect business core concern

AOP Terminologies

- Weaving: It represents a mechanism of associating cross cutting concern to core concern dynamically.
- Proxy: Proxy is an object produced through weaving. There are two types: static proxy and dynamic proxy.
    - In static proxy, static method is used to develop and maintain proxy classes for each business method.
    - In dynamic proxy, proxy is developed at run time.
- Weaver: Code that produces proxies
- Proxy design pattern: Actual object (business object) is wrapped into another object knows as proxy and substitutes that object in place of actual object

We will discuss JoinPoint, Pointcut, and Proxy in this lesson.

AOP Terminologies

Pointcut

Pointcut defines where exactly the Advices have to be applied in various Join Points.

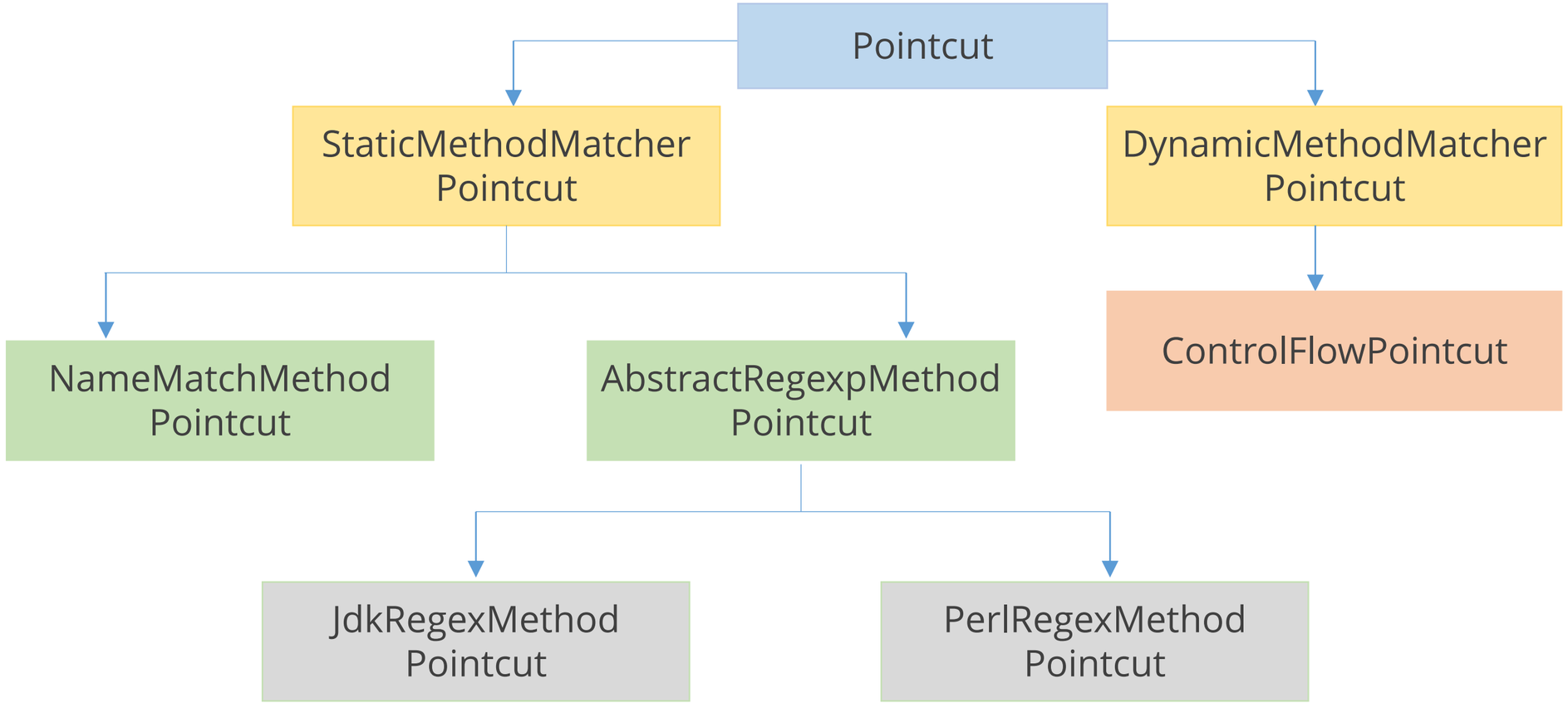
Generally, they act as Filters for the application of various Advices into the real implementation.

AOP Terminologies

Pointcut: TYPES

Springs defines two types of Pointcut:

1. Static: It verifies whether the join point has to be advised or not. It does this once the result is catched @reused.
2. Dynamic: It verifies every time as it has to decide the Join Point based on the argument passes to method call.



AOP Terminologies

StaticMethodMatcherPointcut

StaticMethodMatcherPointcut has two types of Pointcuts:

1. NameMatchMethodPointcut: It interrupts a method via ‘pointcut’ and ‘advisor’

2. AbstractRegexpMethodPointcut: Besides matching method by name, it can match the method’s name by using regular expression pointcut.

   This pointcut is used to verify join point based on pattern of the method name instead of name.

AOP Terminologies

AbstractRegexpMethodPointcut

1. JdkRegexMethod Pointcut: It is represented by org.springframework.aop.JdkRegexpMethodPointcut
2. PerlRegexMethodPointcut: It is represented by org.springframework.aop.support.Perl5RegexpMethodPointcut

AOP Terminologies

DynamicMethodMatcherPointcut

This pointcut is used to verify the context from which business method call is made. If method call is made in specific flow, it will be advice; otherwise, it won't.

AOP Terminologies

POINTCUT INTERFACE

Pointcut is an interface; it is represented by org.springframework.aop.Pointcut.

```
public interface Pointcut
{
ClssFilter getClassFilter()
MethodMatcher getMethodMatcher()
}
```

The getClassFilter() method returns a ClassFilter object which determines whether the classObject argument passed to the matches() method should be considered for giving Advices

AOP Terminologies

Joinpoint

A Joinpoint is a candidate point in the **Program Execution** of the application where an aspect can be plugged in.

This point could be a method being called, an exception being thrown, or even a field being modified.

These are the points where your aspect's code can be inserted into the normal flow of your application to add new behavior.

A pointcut defines the Joinpoints where associated Advice should be applied.

AOP Terminologies

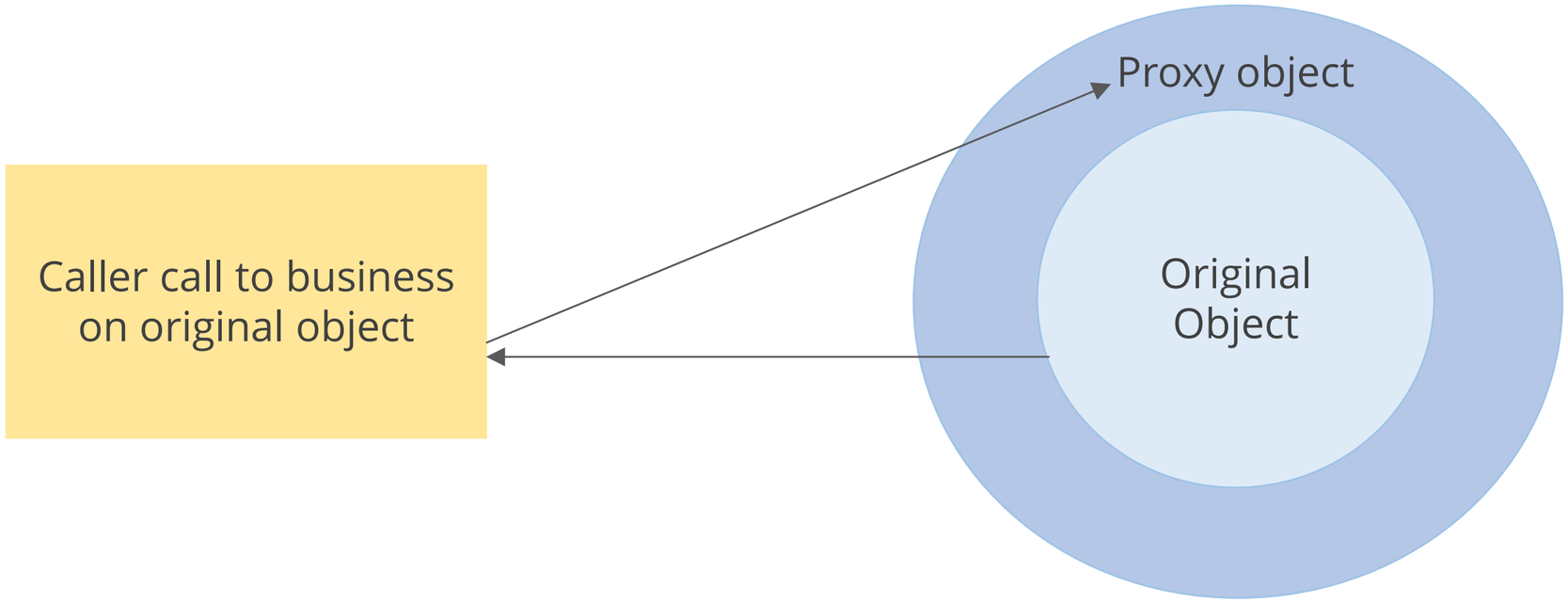
Joinpoint METHODS

Return Type	Method Name	Description
AccessibleObject	getStaticPart()	Return the static part of Joinpoint
Object	getThis()	Return the object that holds the current Joinpoint static part
Object	proceed()	Proceed to the next interceptor in the chain

How Does AOP work?

Proxy interrupts the call made by caller to original object. It will have chance to decide whether and when to pass on the call to original object. In the meantime, any additional code can be executed.

Spring uses the dynamic proxy approach.



A dynamic proxy class is a class that implements a list of interfaces specified at runtime so that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface.

AOP Terminologies

Proxy

- In proxy pattern, a class represents functionality of another class. This type of design pattern is a structural pattern.
- It creates an object that has original object to interface its functionality to outer world. Proxy design pattern is used when we want to provide controlled access to a functionality.
- It is also used to save on the amount of memory used. Similarly, if you want to control access to an object, the pattern becomes useful.

Ways to Generate Dynamic Proxy

1. jdk approach: When the business class implements interface, jdk creates proxy for the business object

2. cglib approach: cglib.jar is used when a business class fails to implement any interface, and jdk is unable to create proxy

AOP Terminologies

Advice

Advice refers to the actual implementation code for an Aspect. Spring supports Method Aspect.

Different types of aspects available in Spring are:

1. Before advice: It executes before a Joinpoint but does not have the ability to prevent execution flow proceeding to the Joinpoint

2. After advice: It is executed after business method call.

3. Around advice: It surrounds a Joinpoint such as a method invocation. It can perform custom behavior before and after method invocation.

4. Throws Advice is executed if actual method throws exception.

AOP Terminologies

Steps to Configure Advice

1. To configure these advices, we have to depend on ProxyFactoryBean. This Bean is used to create Proxy objects for the implementation class along with the Advice implementation.

2. The property ‘proxyInterface’ contains the Interface Name for which the proxy class has to be generated.

3. The ‘interceptorNames’ property takes a list of Advices to be applied to the dynamically generated proxy class.

4. Finally, the implementation class is given in the ‘target’ property.

Before Advice

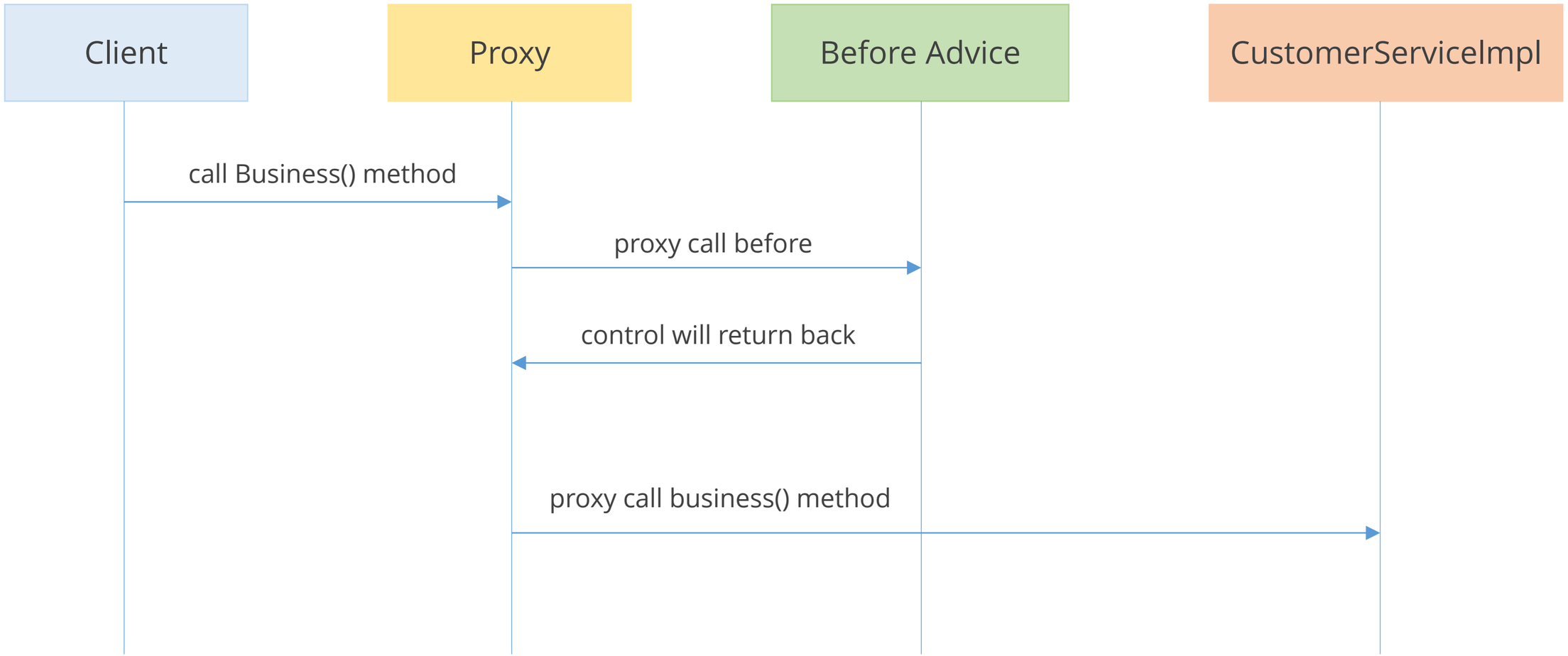
- It is used to intercept before the method execution starts. For example, a system may need to perform some logging operations before allowing users to access resources
- Whenever caller calls the business method, all before advice code is executed prior to the business method.

If exception occurs in the method, afterReturning() method will never be called.

Configuring Before Advice

1) Develop a business interface and its implemented class
2) Develop a class that implements AfterReturningAdvice interface
3) Override its afterReturning() method[(Method method, Object[] args, Object target)]. Cross cutting code is written here.
    - afterReturningMethod method - target method to be invoked
    - Object[] args - various arguments that are passed on to the method
    - Object target - target reference to the object that is calling the method
4) Get a proxy
5) Make a method call

Steps Diagram: Before Advice



After Advice

- It is used to intercept after the method execution. For example, a system needs to perform some delete operations after logging out.

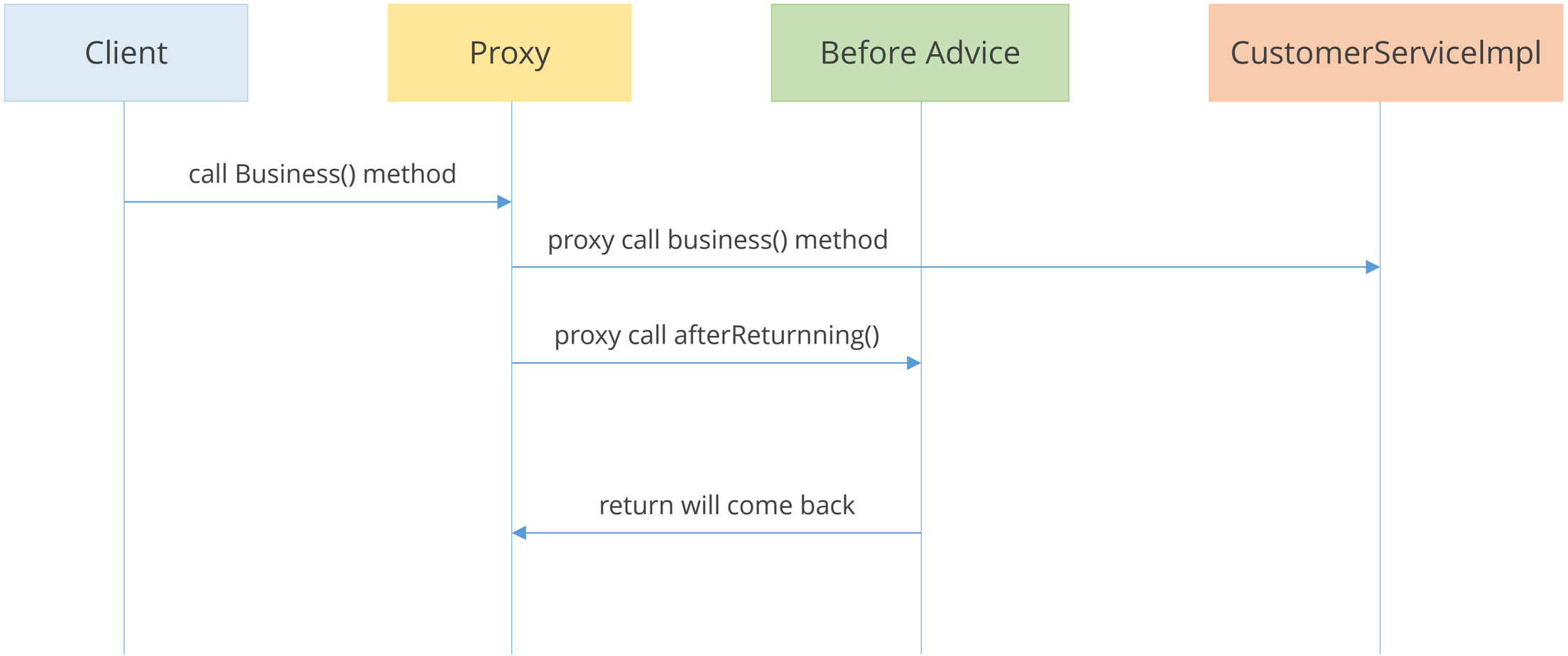
- After Advice is executed after business method.

Configuring After Advice

1. Develop a business interface and its implemented class
2. Develop a class that implements AfterReturningAdvice interface
3. Override its afterReturning (Method method, Object[] args, Object target) method. Cross cutting code is written here.
    - Method method - target method is invoked
    - Object[] args - various arguments that are passed to the method
    - Object target - target reference to the object that is calling the method
4. Get a proxy
5. Make a method call

If exception occur in the method, afterReturning() method will never be called.

Steps Diagram: After Advice



Around Advice

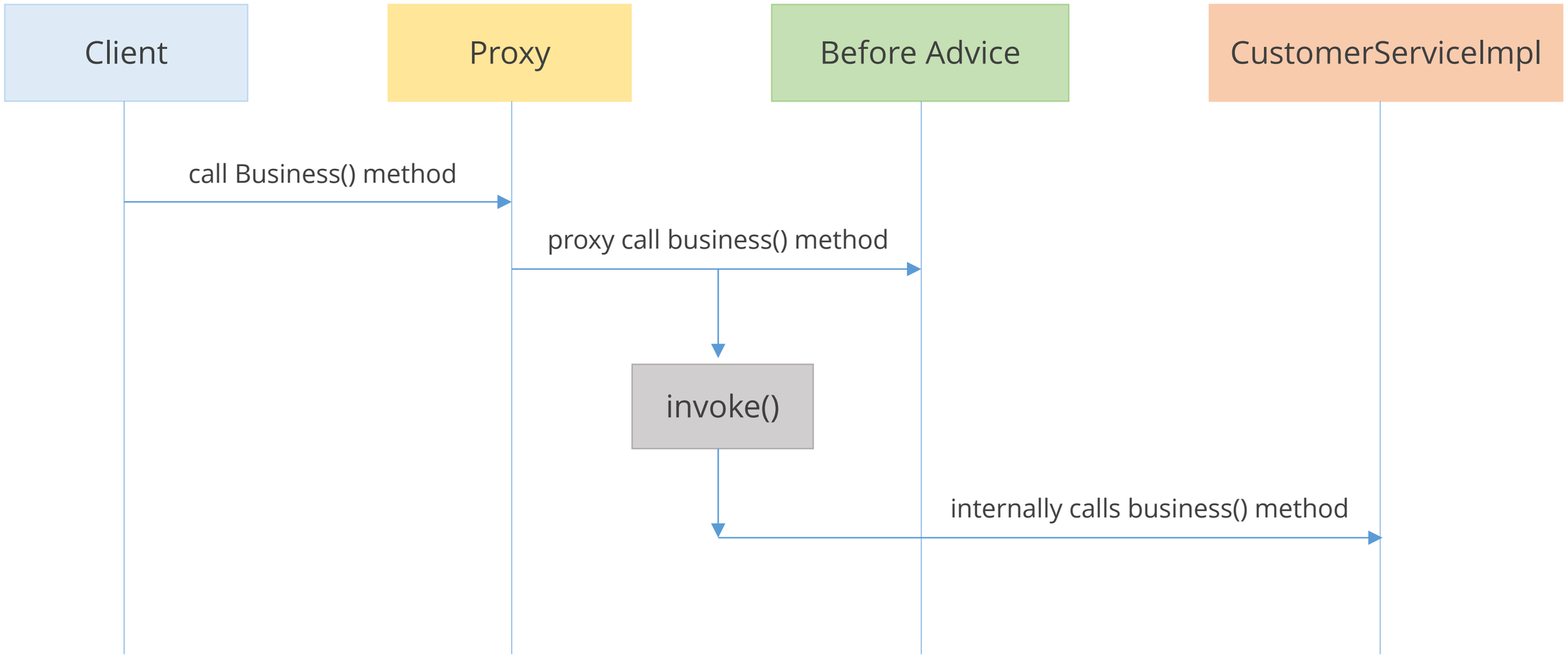
- It can change the return argument of the method call.
- It provides finer control regarding whether the target method has to be called or not.
- It can be performed before business logic or after business logic.

Configuring Around Advice

- Develop a class that implements MethodInterceptor interface
- Override its invoke method. Write cross cutting code
- Get a proxy
- Make a method call
- Parameter of invoke method()

```
public Object invoke(MethodInvocation methodInvocation)
```

Steps Diagram: Around Advice



Throws Advice

When an exception happens during the execution of a method, Throws Advice can be used through the means of org.springframwork.aop.ThrowsAdvice to handle the exception.

- This advice is executed only when exception is raised
- Advice has to implement org.springframework.aop.ThrowsAdvice. It is just a marker interface
- Exception handling code is considered cross cutting code and can be done for entire service layer in Throwsadvice with different kinds of exception parameters.

Configuring Throws Advice

1. Develop a business interface and its implemented class
2. Develop a class that implements ThrowsAdvice interface
3. Override its afterThrowing method. Cross cutting code is written here.
4. Get a proxy
5. Make a method call
6. Throws Advice can take any of the following forms:

```
public void afterThrowing(Exception ex)

public void afterThrowing(Method method,Object[] args,Object target,Exception
exception)
```

Advanced Java

DEMO—Program on Advices

Advanced Java

Topic 3—Configuring AOP in Java Application using AspectJ Approach

AspectJ Based AOP

There are two ways to use Spring AOP AspectJ implementation:

1. By annotation
2. By XML configuration

AspectJ jar files (aspectjrt.jar, aspectjweaver.jar, aspectj.jar, aopalliance.jar) can be downloaded from https://eclipse.org/aspectj/downloads.php

Annotation Based AOP

1. @Before declares the before advice. It is applied before calling the actual method.
2. @After declares the after advice. It is applied after calling the actual method and before returning result.
3. @Around declares the around advice. It is applied before and after calling the actual method.
4. @AfterReturning declares the after returning advice. It is applied after calling the actual method and before returning result. But, you can get the result value in the advice.
5. @AfterThrowing declares the throws advice. It is applied if actual method throws exception.
6. The @Pointcut annotation is used to define the pointcut.
   - @Pointcut("execution(public * *(..))") is applied on all the public methods.
   - @Pointcut("execution(public Operation.*(..))") is applied on all the public methods of Operation class.
   - @Pointcut("execution(* Operation.*(..))") is applied on all the methods of Operation class.
   - @Pointcut("execution(public Employee.set*(..))") is applied on all the public setter methods of Employee class.
   - @Pointcut("execution(* Operation.*(..))")
   - private void doSomething() {}
   - The name of the pointcut expression is doSomething().
7. @Aspect declares the class as aspect

@Before, @After, and @Around will be covered in the scope of this lesson.

@Before

Let's look at Perform and TrackPerformance classes. Assume that Perform class contains actual business methods.

The AspectJ Before Advice is applied before the actual business logic method USING @Before advice.

Perform class

```
public  class Perform{
    public void msgone{System.out.println("msg method invoked");}
    public int msgtwo(){System.out.println("m method invoked");return 2;}
    public int msgthree(){System.out.println("k method invoked");return 3;}
}
```

TrackPerformance class

```
@Aspect
public class TrackPerformance{
    @Pointcut("execution(* Operation.*(..))")
    public void k(){}//pointcut name

    @Before("k()")//applying pointcut on before advice
    public void myadvice(JoinPoint jp)//it is advice (before advice)
    {
        System.out.println("Before advice is called");
           }
}
```

@Before

When object of Perform class is created, the following output is generated:

```
calling msg...
Before advice is called
msg() method invoked
calling m...
Before advice is called
msgone() method invoked
calling k...
Before advice is called
Msgtwo() method invoked
```

@After

For @After annotation, use the same Person class and TrackPerformance.

Person class is the same, but @After annotation in TrackPerformance class is used in this case.

TrackPerformance class

```
@Aspect
public class TrackPerformance{
    @Pointcut("execution(* Operation.*(..))")
    public void k(){}//pointcut name

    @After("k()")//applying pointcut on before advice
    public void myadvice(JoinPoint jp)//it is advice (before advice)
    {
        System.out.println("After advice is called");
           }
}
```

@After("k()")//apply Pointcut on after advice

@After

When object of Person class is created, the following output is generated:

```
calling msg...
msgone() method invoked
After advice is called
calling m...
msgtwo() method invoked
After advice is called
calling k...
msgthree() method invoked
After advice is called
```

@Around

In this example, @Around annotation in TrackPerformance class is used.

```
@Aspect
public class TrackPerformance
{
    @Pointcut("execution(* Operation.*(..))")
    public void abcPointcut(){}

    @Around("abcPointcut()")
    public Object myadvice(ProceedingJoinPoint pjp) throws Throwable
    {
        System.out.println("Additional Concern Before calling actual method");
        Object obj=pjp.proceed();
        System.out.println("Additional Concern After calling actual method");
        return obj;
    }
}
```

@Around

When object of Person class is created, the following output is generated:

```
Additional Concern Before calling actual method
msgone () is invoked
Additional Concern After calling actual method
Additional Concern Before calling actual method
msgtwo() is invoked
Additional Concern After calling actual method
  Additional Concern Before calling actual method
msgthree() is invoked
```

XML Configuration Based AOP

It uses the following XML elements to define advice:

- **aop:before:** It is applied before calling the actual business logic method.
- **aop:after:** It is applied after calling the actual business logic method.
- **aop:after-returning:** It is applied after calling the actual business logic method. It can be used to intercept the return value in advice.
- **aop:around:** It is applied before and after calling the actual business logic method.
- **aop:after-throwing:** It is applied if actual business logic method throws exception.

XML Configuration Based AOP

applicationContext.xml for implementing advice

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
<aop:aspectj-autoproxy />

<bean id="opBean" class="com.javatpoint.Operation">   </bean>
<bean id="trackAspect" class="com.javatpoint.TrackOperation"></bean>

<aop:config>
  <aop:aspect id="myaspect" ref="trackAspect" >
     <!-- @Before -->
     <aop:pointcut id="pointCutBefore"   expression="execution(* Perform.*(..))" />
     <aop:before method="myadvice" pointcut-ref="pointCutBefore" />
  </aop:aspect>
</aop:config>

</beans>
```

```
<!-- @After -->
     <aop:pointcut id="pointCutAfter"    expression="execution(* Perform.*(..))" />
     <aop:after method="myadvice" pointcut-ref="pointCutAfter" />
```

Advice vs. Aspectj

Advice	Aspectj
Spring AOP is best used for application-specific tasks such as security, logging, transactions, etc.	AspectJ contains friendly design-patterns.
This is proxy-based AOP. You can use method-execution Pointcut only.	This supports all Pointcuts.
There can be a little runtime overhead.	There is less runtime overhead than that of Spring AOP.
It needs Spring jar files only.	You need extra build process with AspectJ Compiler or have to setup LTW (load-time weaving).

Key Takeaways

- AOP is used in applications that have cross cutting concerns i.e. a piece of logic or code that is written in multiple classes/layers as per the requirements.
- Springs defines two types of Pointcut:

  Static: It verifies whether the join point has to be advised or not. It does this once the result is catched @reused.

  Dynamic: It verifies every time as it has to decide the Join Point based on the argument passes to method call.
- Spring uses the dynamic proxy approach. A dynamic proxy class is a class that implements a list of interfaces specified at runtime so that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface.

Quiz

QUIZ 1

The Beans in Spring are _____ by default.

a. prototype

b. singleton

c. request

d. session

QUIZ 1

The Beans in Spring are _____ by default.

a. prototype

b. singleton

c. request

d. session

The correct answer is **b. singleton**

The Beans in Spring are singleton by default.

Thank You