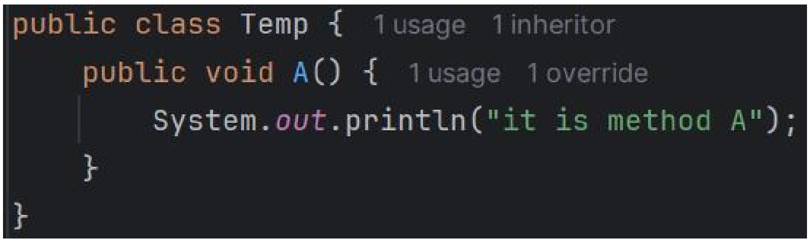
- There are some terminologies in AOP which are **Aspect, Advice, Join Point, Point cut, Proxy** .
- Understanding these terms:
  - Lets assume you have some methods where you want to add some cross-cutting logic.
    - Cross-cutting logic refers to any secondary code apart from business logic.
    - For example: logging, transactions , security etc etc.
  - Some secondary logic needs to be executed before / after / around the execution of these methods.
    - These methods execution is called **Join Points**.
    - Method definition is not Join Point; Method *execution* is **Join Point**.
  - Since many methods can be executed, all these method executions are **Join Points**.
  - You usually do not want to apply cross-cutting logic to all join points, so you need, a filtering logic is required to select some of those.
    - This filtering logic is called a **Pointcut**.
  - A Pointcut defines which join points should be selected for applying cross-cutting logic.
  - The code that has to be executed along with the selected join points, and the timing of its execution, together form an **Advice**
  - Advice defines:
    - What code to execute (cross-cutting logic)
    - When to execute it (e.g., @Before, @After, @Around, etc.)
  - The class that contains all the advice methods is called **Aspect.**

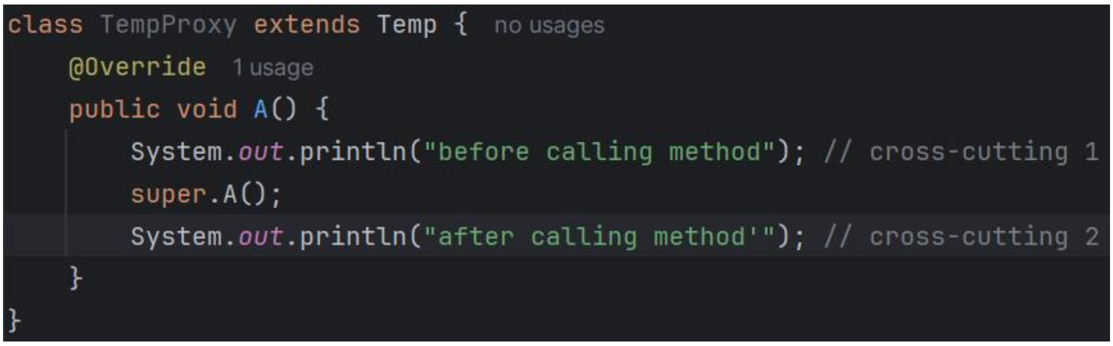
## ➢ Proxy

- In spring, when you create a bean of a class, Spring doesn't assign the object of that exact class; rather it assign an object of the proxy of that class (proxy class extends the real class).
- Lets say this is your real class:



```
public class Temp {  1 usage  1 inheritor
    public void A() {  1 usage  1 override
        System.out.println("it is method A");
    }
}
```

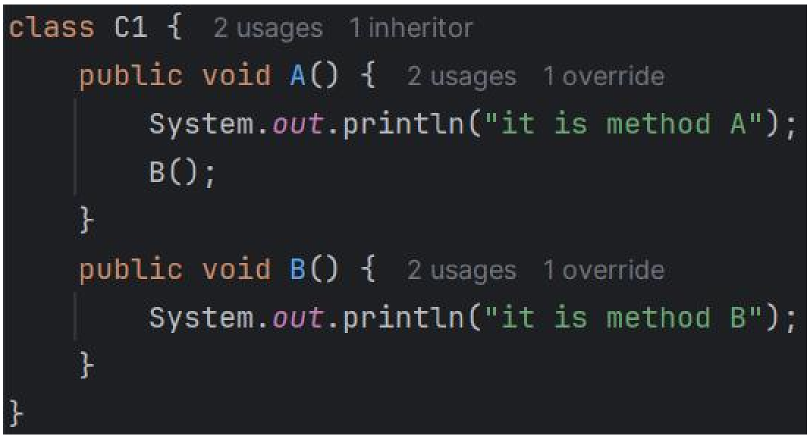
- When you write **@Autowired** or @Configuration, @Bean or anything to get a bean of that class, you'll get a bean of a class of following type:



```
class TempProxy extends Temp {  no usages
    @Override  1 usage
    public void A() {
        System.out.println("before calling method"); // cross-cutting 1
        super.A();
        System.out.println("after calling method'"); // cross-cutting 2
    }
}
```

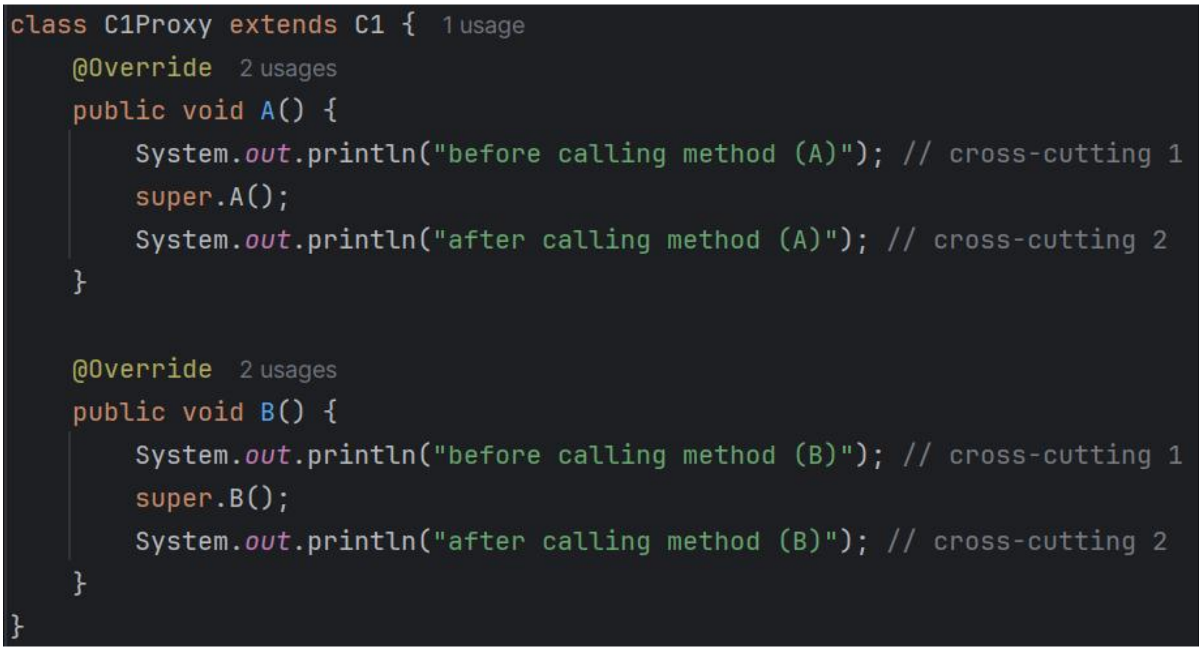
  - This overridden method will be containing all the cross cutting logic and call the real method (its parent class which is real class)

- Consider the below scenario:

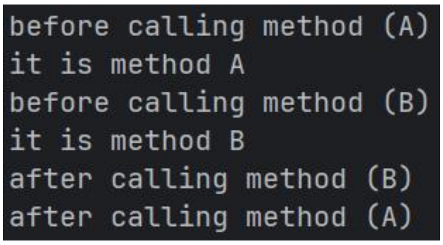


```
class C1 {  2 usages  1 inheritor
    public void A() {  2 usages  1 override
        System.out.println("it is method A");
        B();
    }
    public void B() {  2 usages  1 override
        System.out.println("it is method B");
    }
}
```

  - Here I am calling **B()** inside the method **A()** in the real class.

  - 

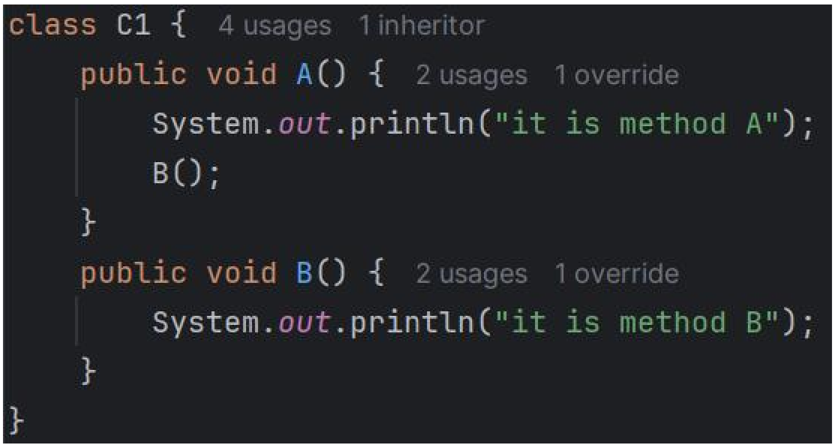
```
class C1Proxy extends C1 {  1 usage
    @Override  2 usages
    public void A() {
        System.out.println("before calling method (A)"); // cross-cutting 1
        super.A();
        System.out.println("after calling method (A)"); // cross-cutting 2
    }

    @Override  2 usages
    public void B() {
        System.out.println("before calling method (B)"); // cross-cutting 1
        super.B();
        System.out.println("after calling method (B)"); // cross-cutting 2
    }
}
```

    - It will be proxy class which object will be assigned to your variable.
  - Now lets say you call **proxyObject.A()** then what will happen? In plain **Java**
    - proxyObject.A() -------------------- C1Proxy's A()
    - super.A() -------------------- C1's A()
    - this.B() --------------------- **C1Proxy's** B()
  - Here the output will be proper:
    - 

```
before calling method (A)
it is method A
before calling method (B)
it is method B
after calling method (B)
after calling method (A)
```

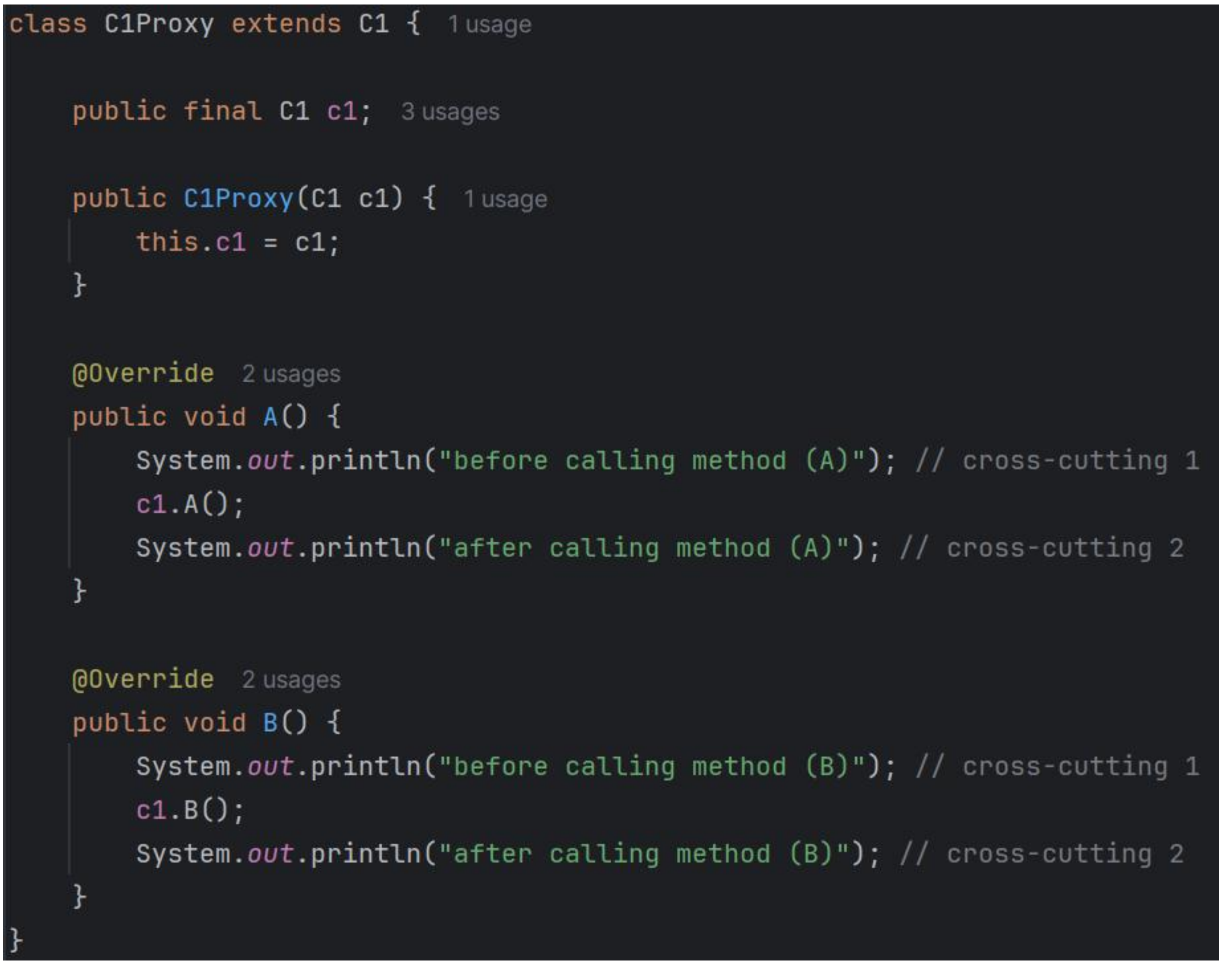
(output)

    - Because here **this** will refer to the object type only which is of type **C1Proxy**.

- In case of Spring AOP, the below happens:
  - 

```
class C1 {  4 usages  1 inheritor
    public void A() {  2 usages  1 override
        System.out.println("it is method A");
        B();
    }
    public void B() {  2 usages  1 override
        System.out.println("it is method B");
    }
}
```

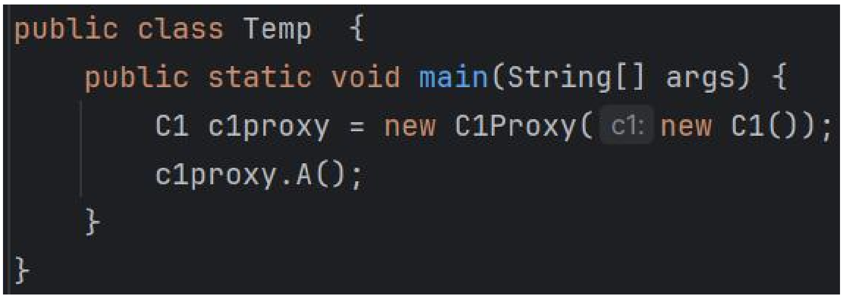
    - It is the real class **C1**.



```java
class C1Proxy extends C1 {  1 usage

    public final C1 c1;  3 usages

    public C1Proxy(C1 c1) {  1 usage
        this.c1 = c1;
    }

    @Override  2 usages
    public void A() {
        System.out.println("before calling method (A)"); // cross-cutting 1
        c1.A();
        System.out.println("after calling method (A)"); // cross-cutting 2
    }

    @Override  2 usages
    public void B() {
        System.out.println("before calling method (B)"); // cross-cutting 1
        c1.B();
        System.out.println("after calling method (B)"); // cross-cutting 2
    }
}
```

- It is the proxy class **C1Proxy**.
- But it doesn't extend the real class, rather it keeps one object of the real class.

Now lets say you call **proxyObject.A()** then what will happen? In plain **Spring AOP**

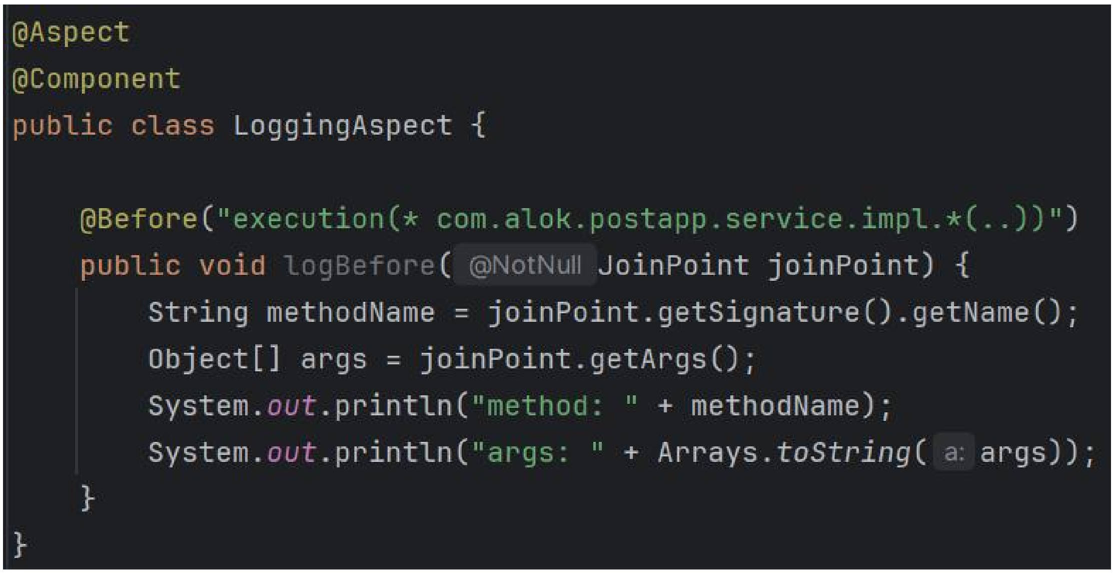


```java
public class Temp  {
    public static void main(String[] args) {
        C1 c1proxy = new C1Proxy( c1: new C1());
        c1proxy.A();
    }
}
```

- c1Proxy.A()  --------------------  C1Proxy's A()
- super.A()  ---------------------  C1's A()
- this.B()  ---------------------  **C1**'s B()
  - Because here *public final C1 c1;* is the real class's object which is **C1**.

- In our code, **Proxy** is a proper class **(TempProxy)** that extends the real class **(Temp)**,
- This is why, calling one method (present inside the same class) from another method will not work in case of AOP.

- The below is a simple template of spring AOP syntax



```
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.alok.postapp.service.impl.*(..))")
    public void logBefore( @NotNull JoinPoint joinPoint) {
        String methodName = joinPoint.getSignature().getName();
        Object[] args = joinPoint.getArgs();
        System.out.println("method:  " + methodName);
        System.out.println("args:  " + Arrays.toString( a: args));
    }
}
```

- LoggingAspect class → **Aspect**
- **@Before** + content of *logBefore* method → **Advice**
- "execution(……)" → **Pointcut**
- Method execution matched by the pointcut → Join Point
- Aspect contains Advice, Advice uses Pointcut, Pointcut selects Join Points

- **JoinPoint and ProceedingJoinPoint**

- You can get an object of type **JoinPoint** to get the details about the method (join point)
- For the advice type **@Around**, you can get **ProceedingJoinPoint** which contains the features of **JoinPoint** + some extra features.
- **JoinPoint** is kind of observer which can get the details about the method and all, but **ProceedingJoinPoint** can control the method execution and all.
- ProceedingJoinPoint is only valid in case of **@Around** advice type.

## ➢ NOTE

➢ Difference between **..** and **.***

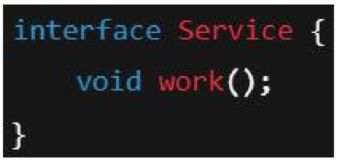
- com.xyz..service.*.*(..) and com.xyz.*.service.*.*(..)
- These both looks same but they are not.
- **com.xyz..service.*.*(..)** means it matches ***any number of sub-package levels***.
  - com.xyz.service
  - com.xyz.*user*.service
  - com.xyz.*user.order*.service
  - com.xyz.*a.b.c*.service
- **com.xyz.*.service.*.*(..)** means it matches ***only one sub-package level***.
  - com.xyz.*user*.service
  - com.xyz.*order*.service
  - com.xyz.*admin*.service
  - com.xyz.*user.order*.service ❌ IT'LL NOT MATCH

# Dynamic Proxies

- **In pure Java, whatever you call from a object, it gets called inside that object only. There is *no involvement of any proxy by default*. But you can create proxy and use it.**

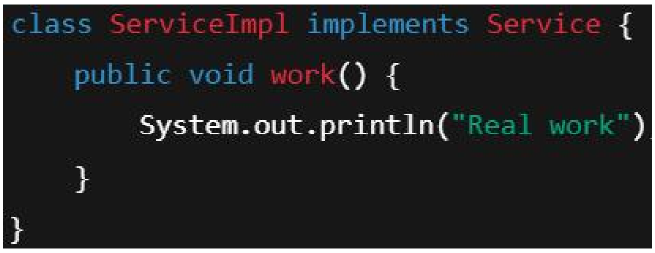
## JDK Dynamic Proxy

- It is a proxy built into **Java itself** that works only with interfaces.
- **No interface → no JDK dynamic proxy**
- Java cannot modify existing classes at run-time, but it can generate a new class implementing the same interface at run-time.
- **Steps to create custom proxy in Pure Java**



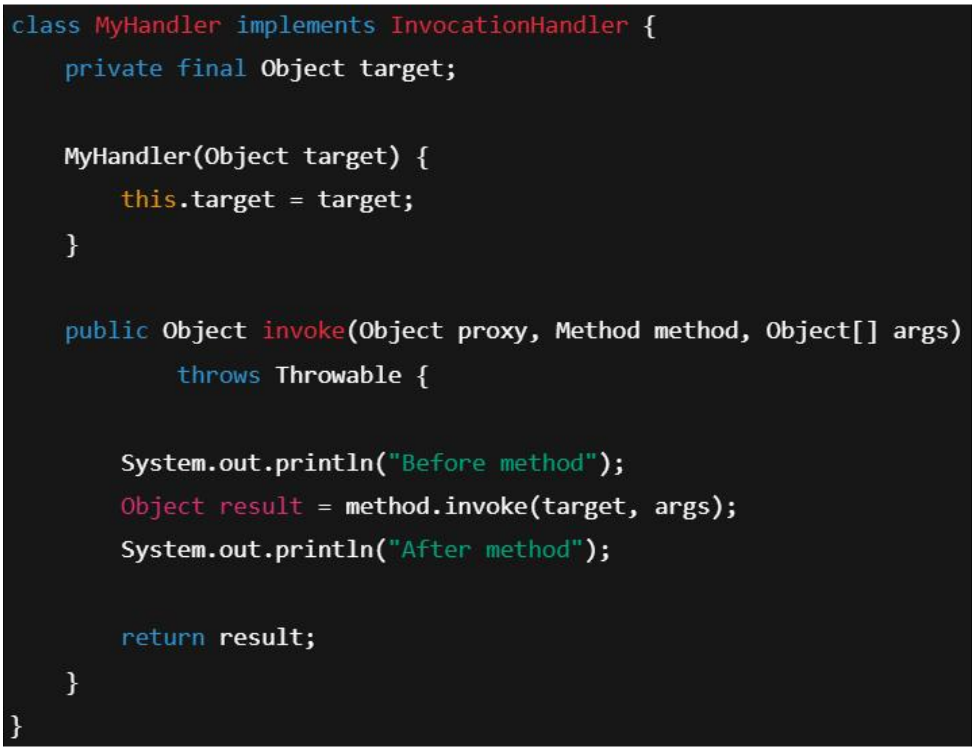
```java
interface Service {
    void work();
}
```

(real interface)



```java
class ServiceImpl implements Service {
    public void work() {
        System.out.println("Real work")
    }
}
```

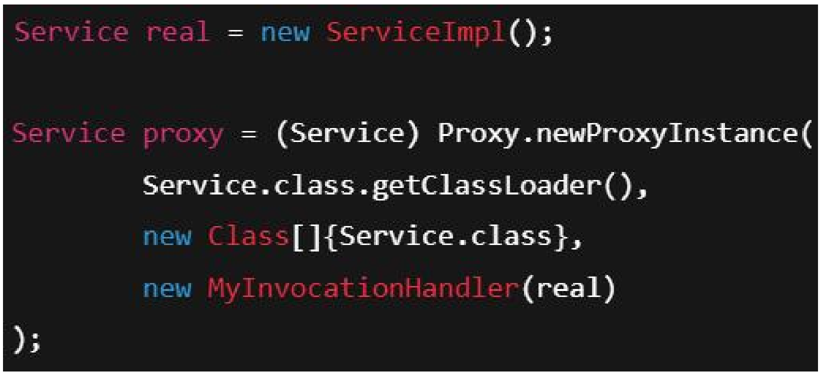
(real class)



```java
class MyHandler implements InvocationHandler {
    private final Object target;

    MyHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {

        System.out.println("Before method");
        Object result = method.invoke(target, args);
        System.out.println("After method");

        return result;
    }
}
```

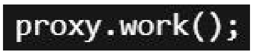
(*self created* handler: it’s the main thing)

- This is where interception happens.
- **method.invoke(target, args)** -- it means call the method inside the **target** with the arguments **args.**

- 

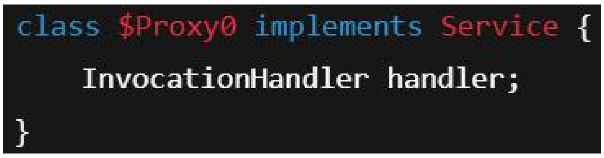
```
Service real = new ServiceImpl();

Service proxy = (Service) Proxy.newProxyInstance(
        Service.class.getClassLoader(),
        new Class[]{Service.class},
        new MyInvocationHandler(real)
);
```

  (create proxy)

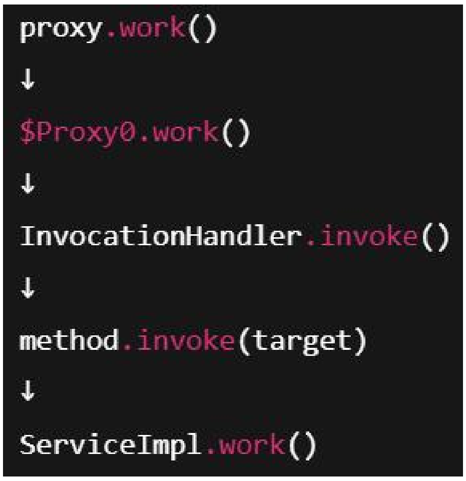
- 

```
proxy.work();
```

  (now call the method)

- At run-time, Java creates something like this:
  - 

```
class $Proxy0 implements Service {
    InvocationHandler handler;
}
```

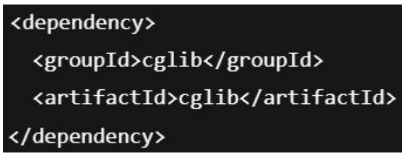
- if you want to get a proxy object then you have to use **Proxy.newProxyInstance** only, if you use **new** keyword to create an object then you'll get a *real object* only, not an proxy object. ----- in Pure Java; not Spring
- This is the flow:
  - 

```
proxy.work()
↓
$Proxy0.work()
↓
InvocationHandler.invoke()
↓
method.invoke(target)
↓
ServiceImpl.work()
```

  - proxy object's method call will lead to InvocationHandler's ***invoke*** method call which do pre & post execution of the actual method call.

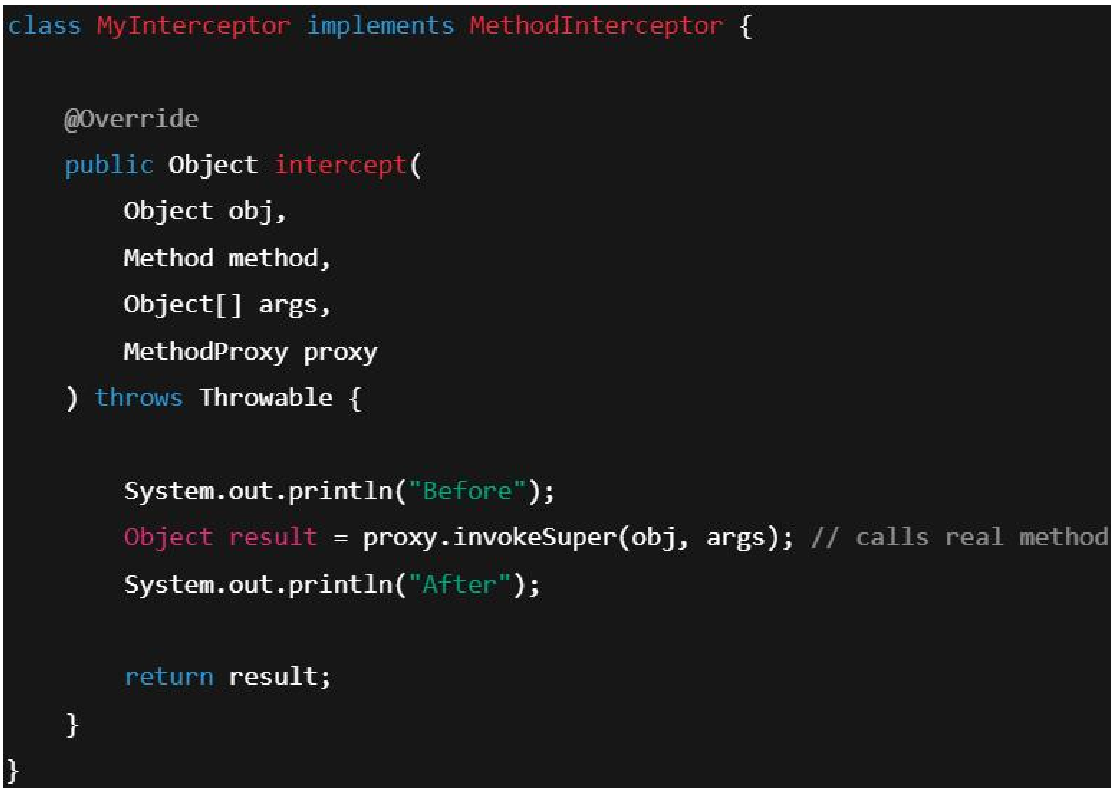
## ➢ CGLIB Proxy

➢ **CGLIB** is not a Pure Java feature; it works if you add the CGLIB library or use any framework like Spring, Hibernate …etc.



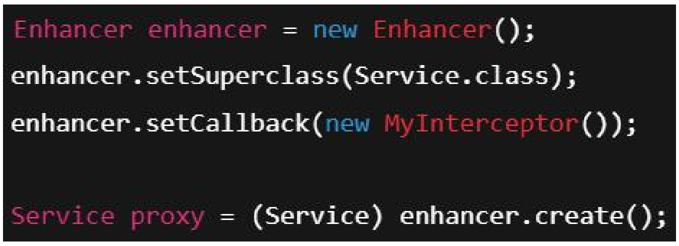
```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
</dependency>
```

- It works even if there is **no interface involved**.
- Steps:
  - Create one interceptor class



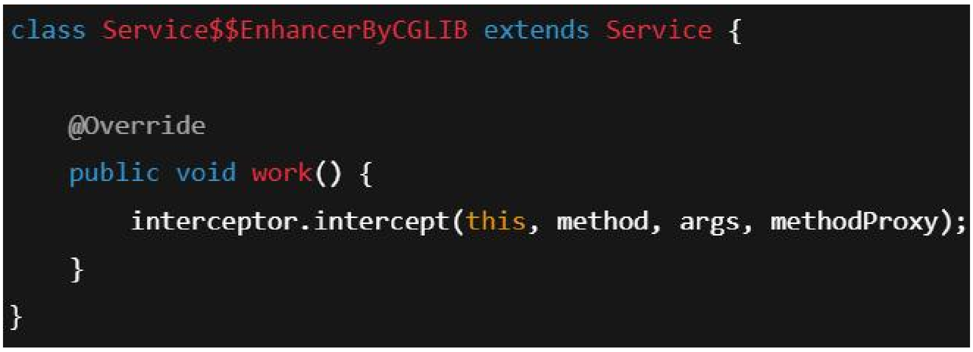
```
class MyInterceptor implements MethodInterceptor {

    @Override
    public Object intercept(
        Object obj,
        Method method,
        Object[] args,
        MethodProxy proxy
    ) throws Throwable {

        System.out.println("Before");
        Object result = proxy.invokeSuper(obj, args); // calls real method
        System.out.println("After");

        return result;
    }
}
```

  - Create CGLIB proxy



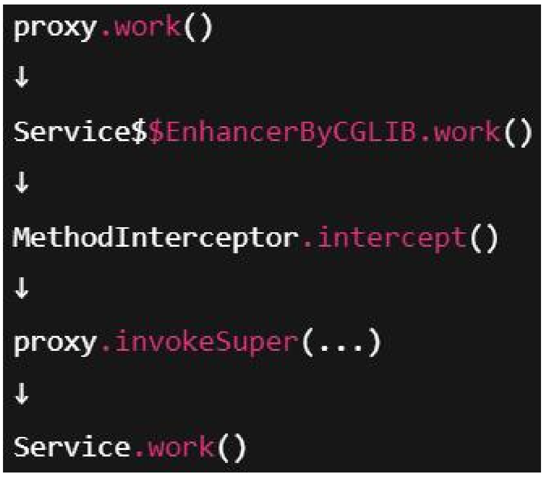
```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Service.class);
enhancer.setCallback(new MyInterceptor());

Service proxy = (Service) enhancer.create();
```

  - In run-time, CGLIB will create something like this:



```
class Service$$EnhancerByCGLIB extends Service {

    @Override
    public void work() {
        interceptor.intercept(this, method, args, methodProxy);
    }
}
```

- When you call **proxy.work()** the below flow happens



```
proxy.work()
↓
Service$$EnhancerByCGLIB.work()
↓
MethodInterceptor.intercept()
↓
proxy.invokeSuper(...)
↓
Service.work()
```

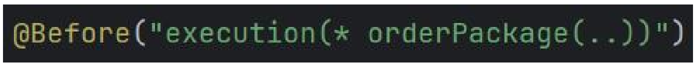
## ➢ Proxy used by Spring AOP

➢ Spring AOP uses **JDK Dynamic Proxy** *if an interface exists*; otherwise it uses CGLIB (or always CGLIB if forced).

# Pointcut

## ➢ execution pointcut

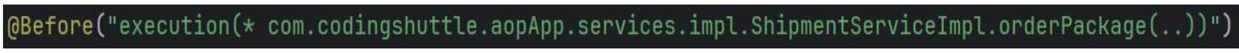
- ➢ If you don’t give any reference to the method in the *Pointcut* statement, then it’ll select all the method having that name in all the directories.

  - 

    ```
    @Before("execution(* orderPackage(..))")
    ```

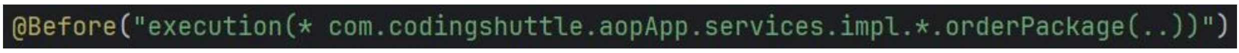
  - Now, all the methods present having name *orderPackage()* throughout the directories, will be selected.

- ➢ If you write the full reference then only that specific method present in that reference will be selected.

  - 

    ```
    @Before("execution(* com.codingshuttle.aopApp.services.impl.ShipmentServiceImpl.orderPackage(..))")
    ```

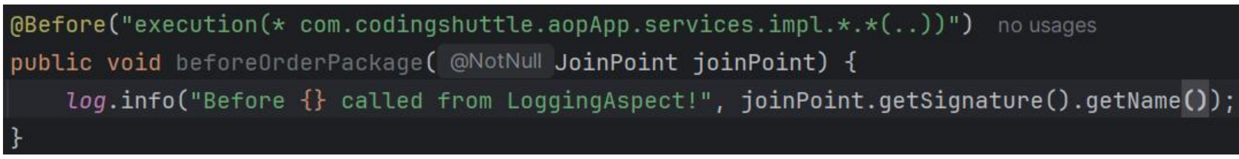
- ➢ You can also loose the reference by giving * in between.

  - 

    ```
    @Before("execution(* com.codingshuttle.aopApp.services.impl.*.orderPackage(..))")
    ```

  - All the *orderPackage()* methods inside all the classes inside *impl* package will be selected.

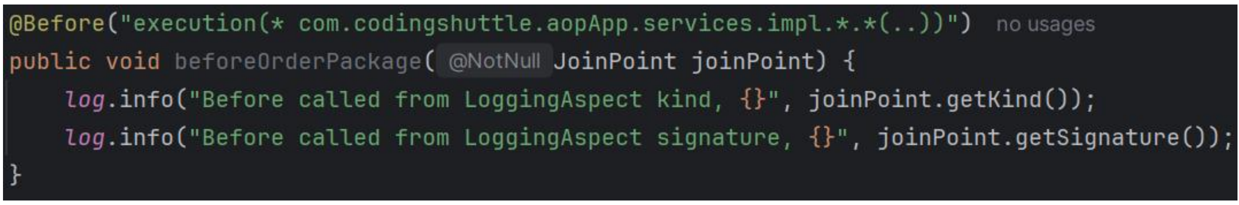
- ➢ See the below:

  - 

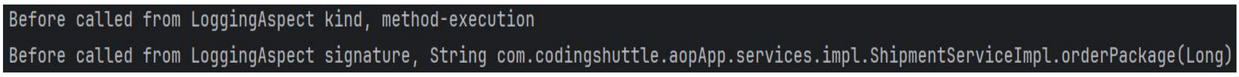
    ```
    @Before("execution(* com.codingshuttle.aopApp.services.impl.*.*(..))")  no usages
    public void beforeOrderPackage( @NotNull JoinPoint joinPoint) {
        log.info("Before {} called from LoggingAspect!", joinPoint.getSignature().getName());
    }
    ```

  - Here all method present inside all the classes present inside *impl* package will be selected.

- ➢ You can also get the *kind* of join point

  - 

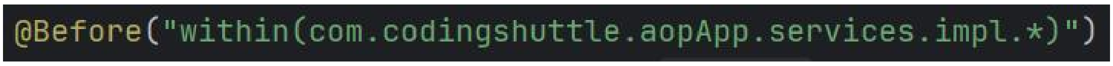
    ```
    @Before("execution(* com.codingshuttle.aopApp.services.impl.*.*(..))")  no usages
    public void beforeOrderPackage( @NotNull JoinPoint joinPoint) {
        log.info("Before called from LoggingAspect kind, {}", joinPoint.getKind());
        log.info("Before called from LoggingAspect signature, {}", joinPoint.getSignature());
    }
    ```

  - 

    ```
    Before called from LoggingAspect kind, method-execution
    Before called from LoggingAspect signature, String com.codingshuttle.aopApp.services.impl.ShipmentServiceImpl.orderPackage(Long)
    ```

## ➢ **within** pointcut

- Description
  - Use **within** when you want to limit the advice to a particular class or package, without focusing on specific methods.
  - This pointcut applies to any join point within the a package like *com.example.aopApp.services* package, including **methods, fields, and constructors**.

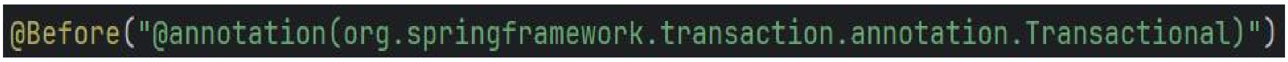


```
@Before("within(com.codingshuttle.aopApp.services.impl.*)")
```

  - It'll apply to all the classes present inside *impl* package.

## ➢ **@annotation** pointcut

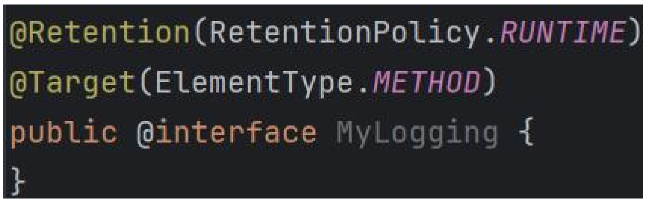
- It is used to apply advice to methods annotated with a particular annotation.
- You can also create your own custom annotation and apply advice to the methods which use your custom annotation.



```
@Before("@annotation(org.springframework.transaction.annotation.Transactional)")
```

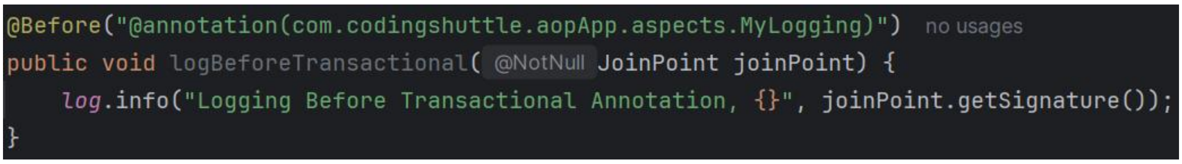
  - It'll apply advice to all the methods annotated with **@Transactional** annotation.

- Using custom annotation for advice
  - I created one custom logging annotation:



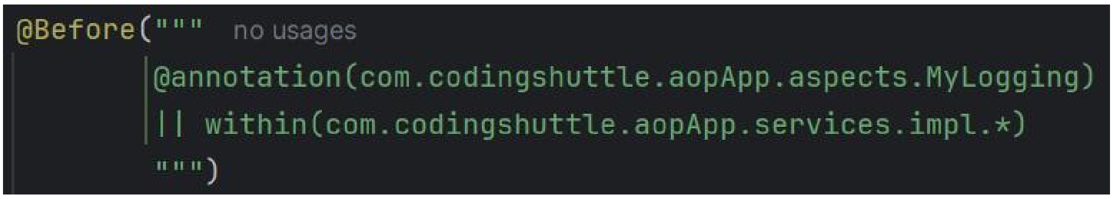
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyLogging {
}
```

  - Now use this annotation in the pointcut

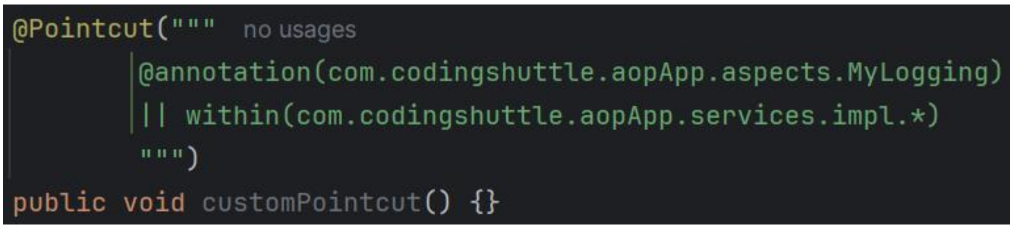


```
@Before("@annotation(com.codingshuttle.aopApp.aspects.MyLogging)")  no usages
public void logBeforeTransactional( @NotNull JoinPoint joinPoint) {
    log.info("Logging Before Transactional Annotation, {}", joinPoint.getSignature());
}
```

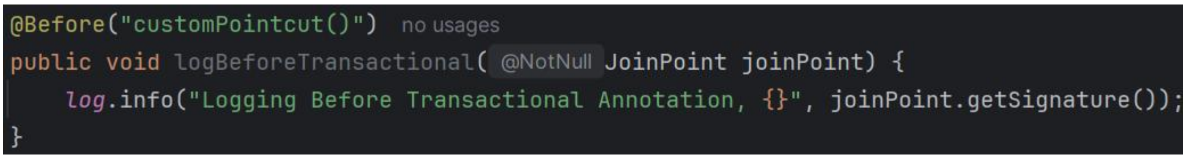
- Also you can combine multiple pointcuts with **&&** or **||** operators.



```
@Before("""  no usages
        @annotation(com.codingshuttle.aopApp.aspects.MyLogging)
        || within(com.codingshuttle.aopApp.services.impl.*)
        """)
```

- **Using @Pointcut to declare a pointcut expression and re-use that multiple times.**
  - I created a pointcut rule
    - 

```
@Pointcut("""  no usages
        @annotation(com.codingshuttle.aopApp.aspects.MyLogging)
        || within(com.codingshuttle.aopApp.services.impl.*)
        """)
public void customPointcut() {}
```

  - Now just use it like this
    - 

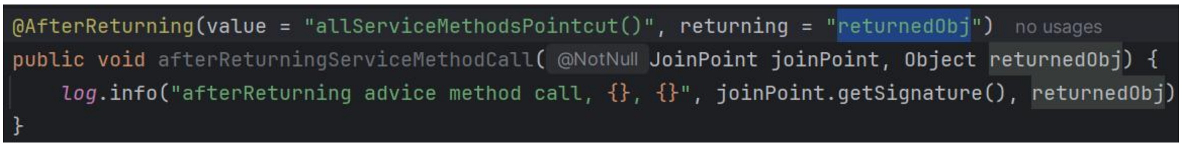
```
@Before("customPointcut()")  no usages
public void logBeforeTransactional( @NotNull JoinPoint joinPoint) {
    log.info("Logging Before Transactional Annotation, {}", joinPoint.getSignature());
}
```

    - If you want to apply multiple types of advices like @Before, @After ..etc to a same pointcut expression, then it is useful, because you don't need to write it again and again.
  - **@Pointcut methods are never executed; they only name a pointcut expression used by advice.**
    - Even if you write something in that pointcut method (customPointcut() in our case), it'll never be executed.

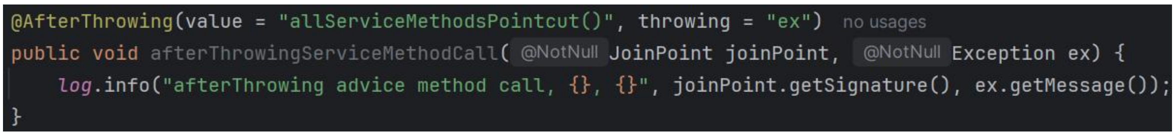
# Advice

- Advice is associated with a pointcut expression and runs **before**, **after**, or **around** method executions matched by the pointcut

  The pointcut expression may be either an inline pointcut or a reference to a named pointcut.

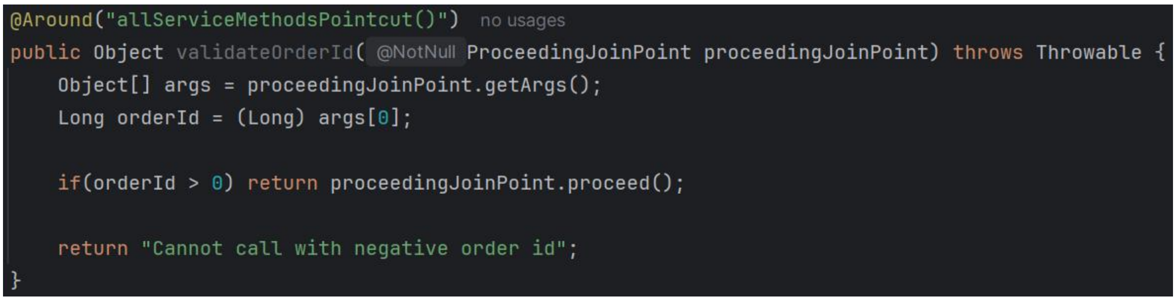
- There are the following advice types
  - **@Before**
  - **@After**
  - **@AfterReturning**
    - 

      ```
      @AfterReturning(value = "allServiceMethodsPointcut()", returning = "returnedObj")  no usages
      public void afterReturningServiceMethodCall( @NotNull JoinPoint joinPoint, Object returnedObj) {
          log.info("afterReturning advice method call, {}, {}", joinPoint.getSignature(), returnedObj)
      }
      ```

    - If you want to get the returned object, then use like this.
    - The name against **returning** should be same as the ***argument name*** inside the advice method. Argument should be of type ***Object***.
  - **@AfterThrowing**
    - 

      ```
      @AfterThrowing(value = "allServiceMethodsPointcut()", throwing = "ex")  no usages
      public void afterThrowingServiceMethodCall( @NotNull JoinPoint joinPoint,  @NotNull Exception ex) {
          log.info("afterThrowing advice method call, {}, {}", joinPoint.getSignature(), ex.getMessage());
      }
      ```

    - Here also, if you want to get the thrown exception, then use like this.
    - The name against **throwing** should be same as the ***argument name*** for exception inside the advice method.
  - **@Around**
    - It has complete control over the method, either to run that, or do something before or after that.

      ```
      @Around("allServiceMethodsPointcut()")  no usages
      public Object validateOrderId( @NotNull ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
          Object[] args = proceedingJoinPoint.getArgs();
          Long orderId = (Long) args[0];

          if(orderId > 0) return proceedingJoinPoint.proceed();

          return "Cannot call with negative order id";
      }
      ```

      - You can do validation in this as well.
      - Use **ProceedingJoinPoint** in case of **@Around** to get more control.

# Spring Proxy & Internal Working of AOP

- **Spring Proxy**
  - Proxies are created only when certain conditions are met, such as when
    - AOP is enabled, and there are pointcuts matching specific methods.
    - Spring features like **@Transactional**, **@Cacheable**, and **@Async** are used, which internally rely on proxies to apply cross-cutting concerns like transaction management, caching, or asynchronous method execution.
  - If a Spring bean doesn't need any cross-cutting functionality (like AOP, transaction management, caching or async processing), no proxy will be created. Spring optimizes its proxy creation to avoid unnecessary overhead when proxies are not needed.
- **How Proxies are managed?**
  - Proxies are created and managed by the Spring container (the ApplicationContext).

    When Spring determines that a bean needs to be proxied (due to AOP advice, transaction management, caching, etc..), it creates a proxy for the bean instead of instantiating the bean directly.
    - This proxy wraps around the actual bean and adds additional behaviour (advice) before and after the actual method calls.
    - Thus, proxies are stored in place of the actual beans in the ApplicationContext.

      Whenever a client retrieves a bean (e.g., via **@Autowired** or **context.getBean()**), what they receive is the proxy object, if proxying is enabled for that bean.
- **How Proxy are created?**
  - Spring AOP typically relies on dynamic proxies (**JDK Dynamic Proxies** or **CGLIB Proxies**) to implement the interception of method calls.
    - **JDK Dynamic Proxies**

      If the target object implements one or more interfaces, Spring uses JDK Dynamic Proxies.

      The proxy is created at runtime, implementing the same interfaces as the target object.

        - **CGLIB Proxies** (*Code Generation Library* Proxies)

          If the target object doesn't implement any interfaces, Spring uses CGLIB, which generates a subclass of the target class at runtime to create the proxy.

    - When a method on the proxy object is called, the proxy intercepts the method call and applies the advice around it.

- **How Proxies are used?**
    - When Spring starts up, it scans the classes for aspects and applies the advice to the relevant beans based on the pointcuts.
    - For each bean, Spring determines whether it needs to be proxied (based on whether any methods match a pointcut).
    - If proxying is needed, Spring creates a proxy object that wraps the original bean.
    - When a client calls a method on the proxied bean, the call is intercepted by the proxy.

      The proxy determines if there is any advice that should be applied based on the pointcuts.
    - If the method does not match any pointcuts, the proxy delegates the call directly to the target object without applying any advice.
    - If advice exists, the proxy executes the advice (e.g., logging, transaction management) before, after, or around the actual method invocation.
    - The most powerful type of advice is **@Around**.

      It allows you to control the execution of the target method by calling

      **ProceedingJoinPoint.proceed()**

      You can choose to execute code before and after the target method execution or even decide not to execute it at all.
    - After the advice has been executed, control is returned to the original method.
    - Any results returned by the method or exceptions thrown are then passed back to the caller.

- **Weaving**
    - In addition to dynamic proxies, Spring AOP can use a process called **weaving** for bytecode manipulation.

      Weaving is the process of injecting aspects into the target class bytecode at different points in the lifecycle.
        - Compile-time weaving:

          Aspects are woven into the code during the compilation process.

    - Load-time weaving:

      Aspects are woven when the class is loaded into the JVM.

    - Post-compilation weaving:

      Aspects are woven after the classes have been compiled.

  - Spring primarily uses dynamic proxies, but with **AspectJ** integration, load-time weaving or compile-time weaving can also be used for more complex scenarios.

- F