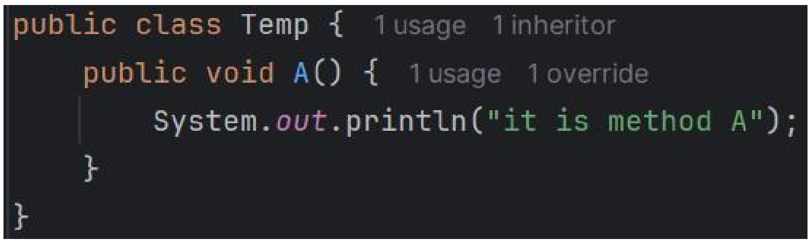
- Role is the main concept for **authorization**. Depending upon the role of the user, it'll be decided that what are the thing he/she can access.
- There are some terminologies in AOP which are **Aspect, Advice, Join Point, Point cut, Proxy** .
- Understanding these terms:
  - Lets assume you have some 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

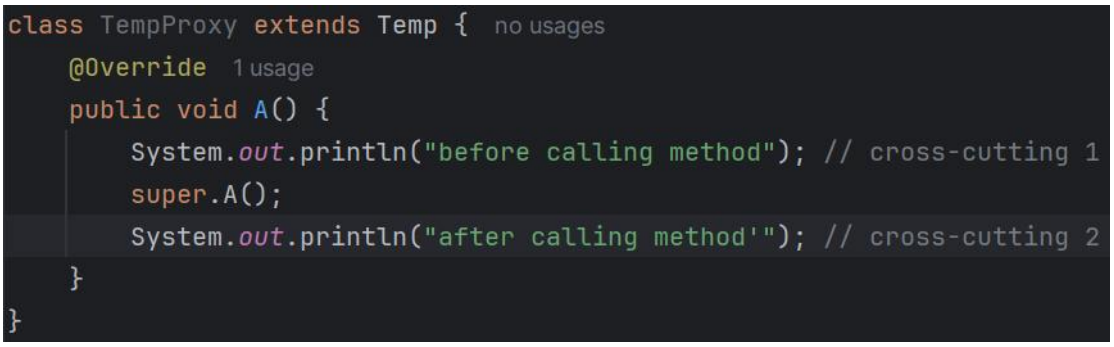
➢ In spring, when you create a bean of a class, Spring doesn't assign the object of that exact class; rather it assign an object of the proxy of that class (proxy class extends the real class).

➢ 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");
    }
}
```

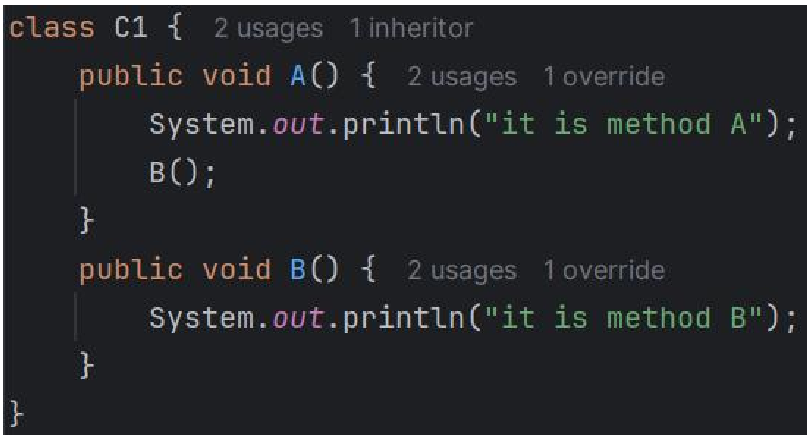
➢ When you write **@Autowired** or @Configuration, @Bean or anything to get a bean of that class, you'll get a bean of a class of following type:



```
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
    }
}
```

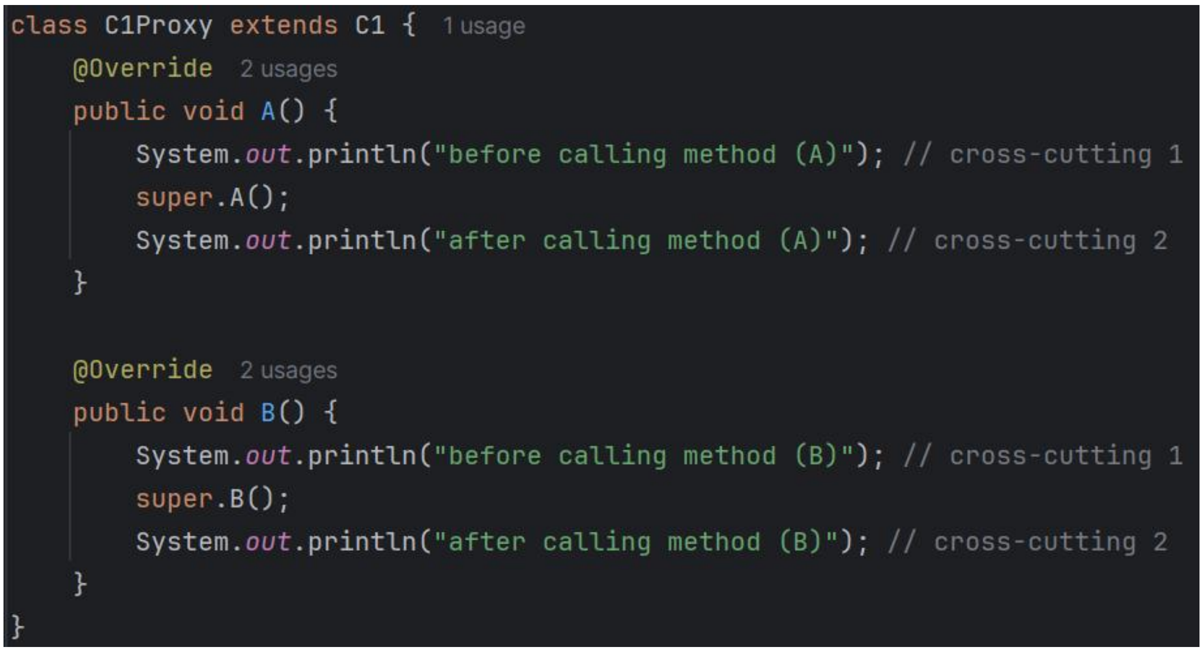
- This overridden method will be containing all the cross cutting logic and call the real method (its parent class which is real class)

➢ 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");
    }
}
```

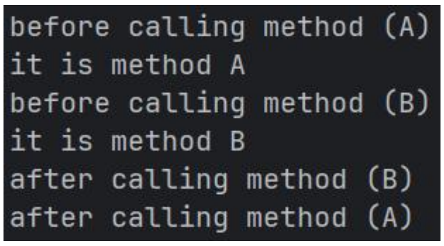
- Here I am calling **B()** inside the method **A()** in the real class.



```java
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()

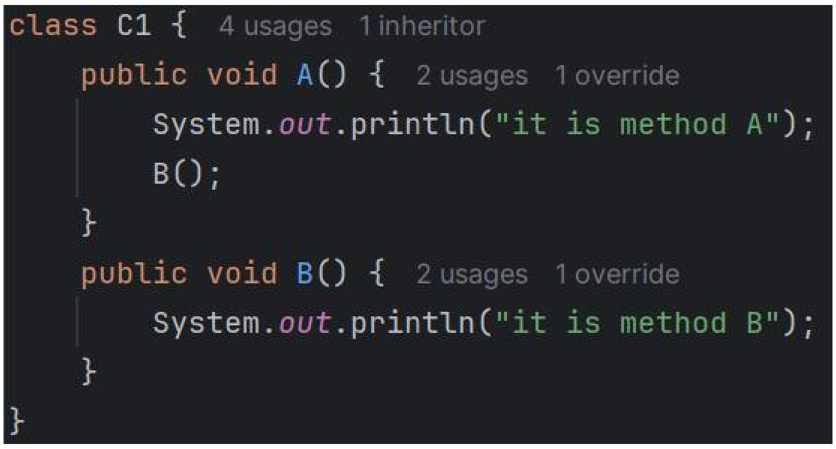
- Here the output will be proper:
  - 

```
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)

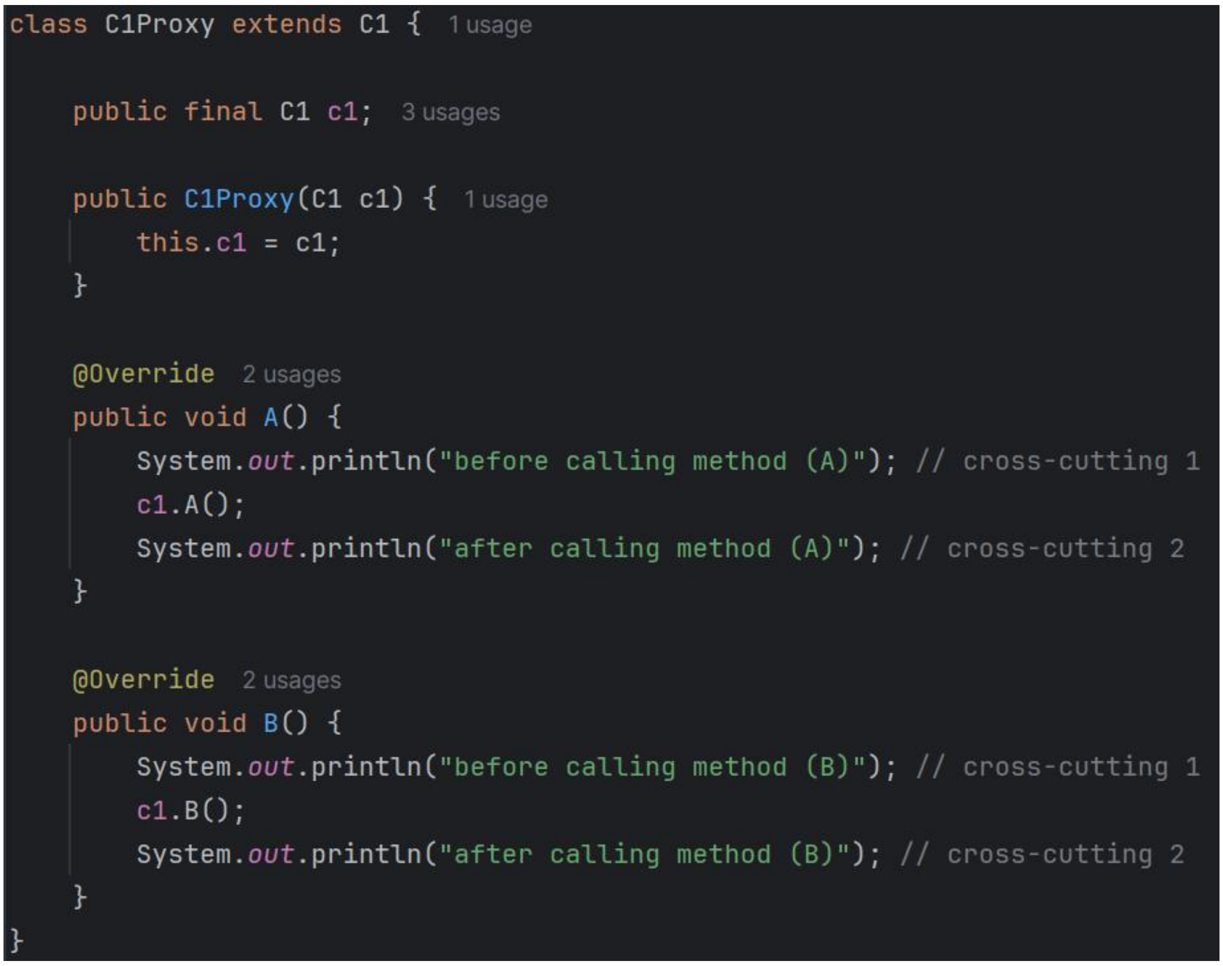
  - Because here **this** will refer to the object type only which is of type **C1Proxy**.

➢ In case of Spring AOP, the below happens:



```java
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");
    }
}
```

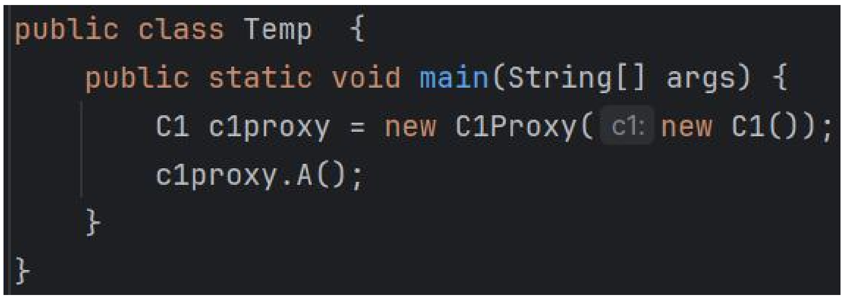
- It is the real class **C1**.



```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**

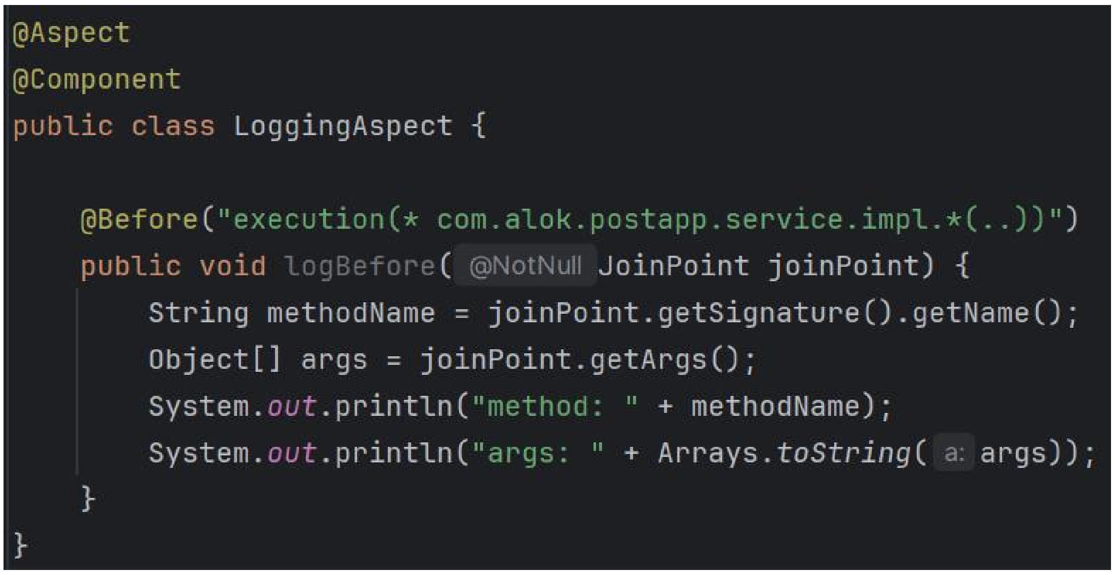


```java
public class Temp  {
    public static void main(String[] args) {
        C1 c1proxy = new C1Proxy( c1: new C1());
        c1proxy.A();
    }
}
```

- 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



```java
@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.

➢

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢

➢ Spring AOP

➢

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

➢ F

م