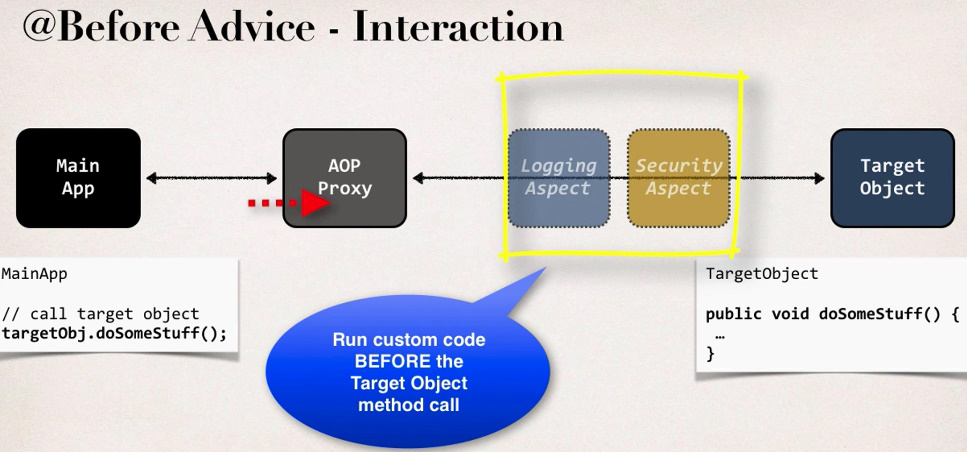
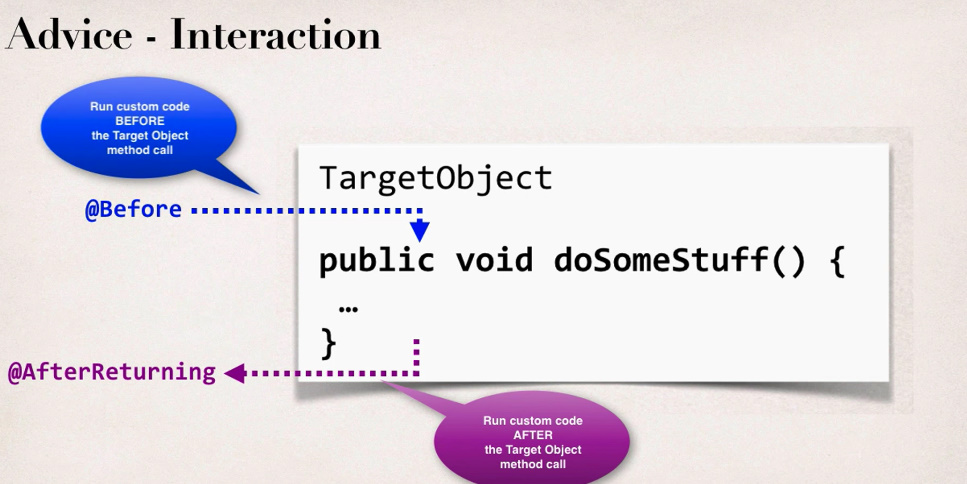
**Ce face**

* Before advice e folosita cand vrem ca inainte ca metoda sa execute instructiunile sa fie executat alt code
* 

Deci, in MainApp avem un obiect targetObj, care ruleaza metoda doSomeStuff() din, care e ceva vreun obiect folosit prin proiect. Totusi, noi vrem ca inainte ca metoda obiectului targetObj din MainApp sa fie executata sa fie executat un alt cod creat de noi, pentru loggin sau security.

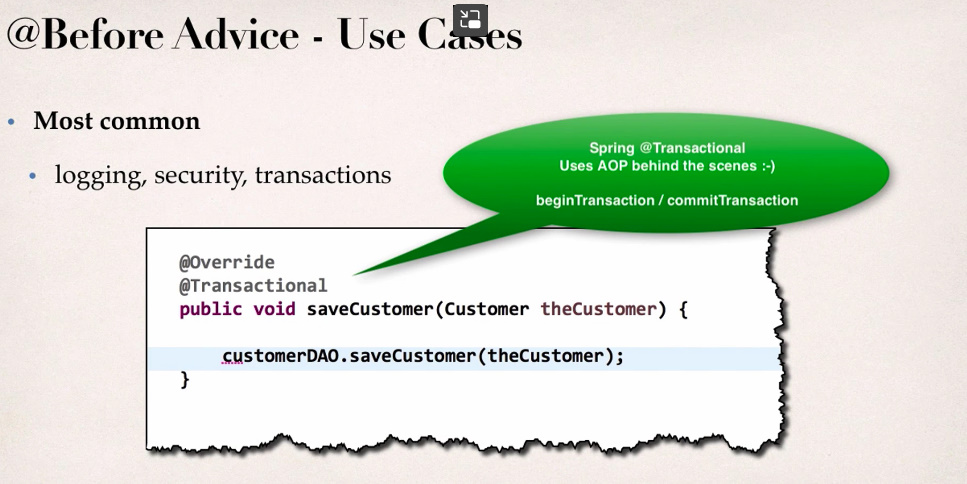
* Pentru asta facem asa:



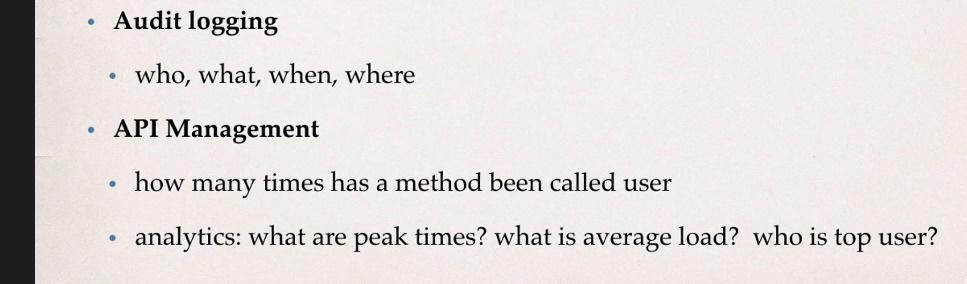
@Before asigura ca atunci cand un obiect de tip TargetObject va rula metoda doSomeStuff, inainte de executia ei sa se execute un alt code

@AfterReturning face ca dupa ce metoda se executa cu succes si ajunge la return sau pur si simplu la final, sa se execute un cod.

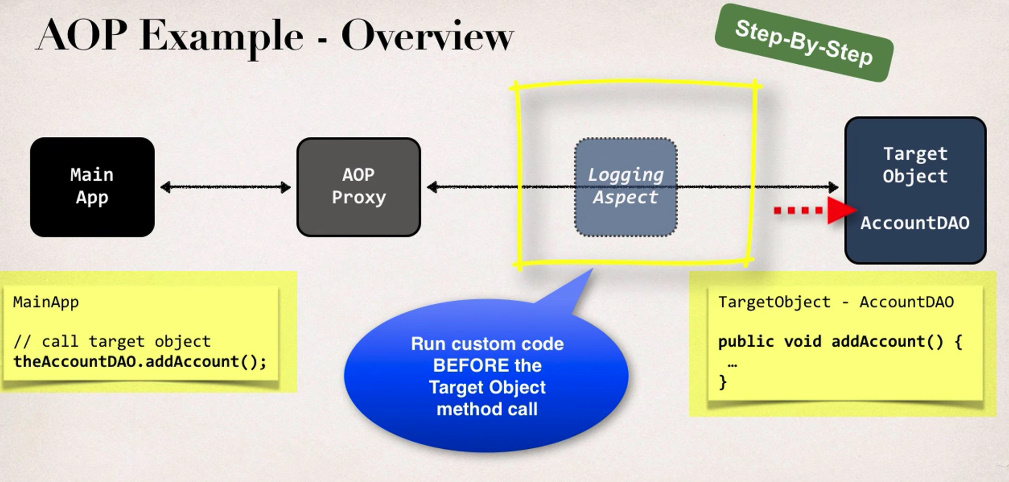
**Unde se foloseste**



@Transactional foloseste AOP pentru inceperea si comiterea tranzactiei.

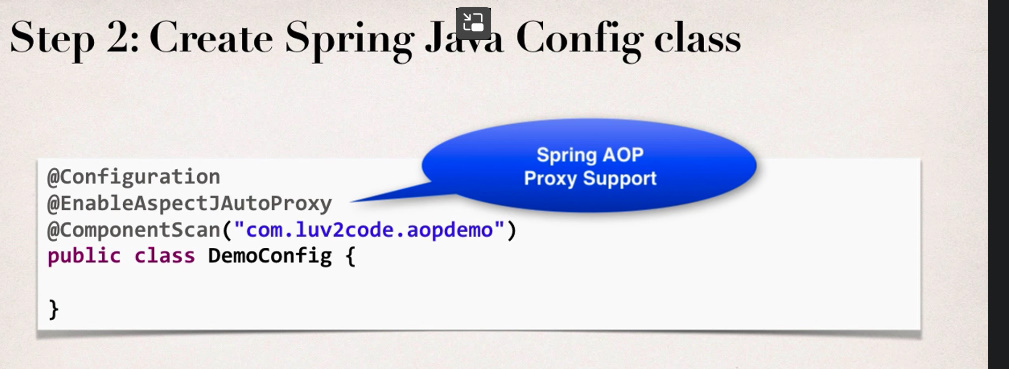


Exemplu concret:



**Cum lucram cu AOP in Project**

* Chiar daca lucram cu Spring AOP, oricum avem nevoie de AspectJ
* 
* Cand vom crea application context cu class, avem nevoie de o noua anotatie:



Ea ne permite sa folosim proxying cand folosim Spring AOP cu AspectJ

E pentru Spring AOP Proxy Support. Defapt, anume ea permite folosirea anotatiei @Aspect

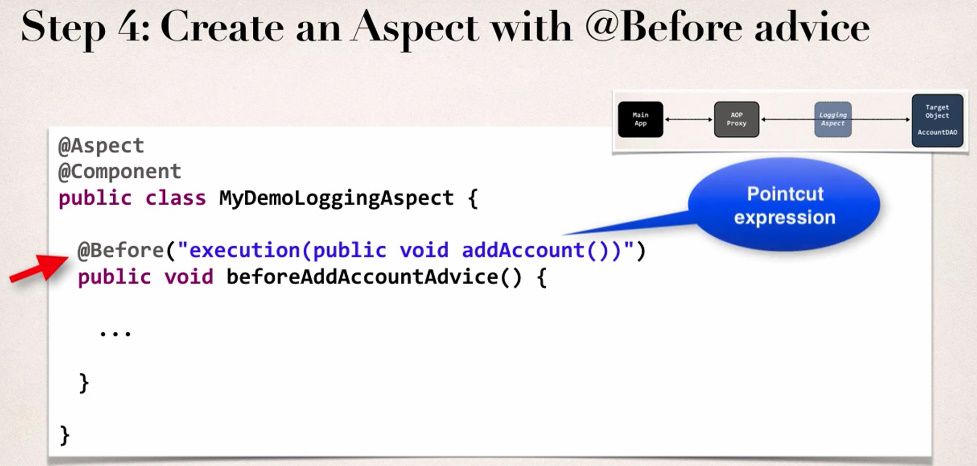
* Un aspect il cream asa:



Un aspect tot trebuie sa fie un bean, de aia are @Component

@Aspect spune lui Spring ca clasa data este un Aspect, si poate fi ca un “spion”, comunicand intre metode.

* Acum trebuie sa specificam in acest aspect inainte la ce metode sa fie executat:





@Before(“execution(accesModifier returnType numeMetoda(param1,...,))”)

* Anume @Before este Advice si in el este PointCut expression, deoarece arata unde se executa codul mai jos de el, si prin unde se are in vedere inaintea carei metode.
* anotatia este Advice, deoarece ea arata unde se fac actiunile necesare
* **Aspectele nu trebuie sa faca operatii prea mari si grele!!**

**Creare proiect**

1. Pentru a adauga AspectJ, folosim dependenta:
2. <dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjrt</artifactId>  
    <version>1.9.19</version>  
   </dependency>  
   <dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjweaver</artifactId>  
    <version>1.9.19</version>  
   </dependency>
3. Cream o clasa Component pentru obiectul target simpla:

@Component  
public class AccountDAO {  
 public void addAccount(){  
 System.*out*.println(getClass() + ": DOING MY DB WORK: ADDING AN ACCOUNT");  
 }  
}

1. Cream app config

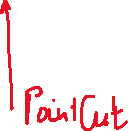
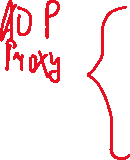
@Configuration  
@EnableAspectJAutoProxy  
@ComponentScan("com.java.org")  
public class DemoConfig {  
}

1. Cream main app

public class MainDemoApp {  
 public static void main(String[] args){  
  
 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(DemoConfig.class);  
  
 AccountDAO accountDAO = context.getBean("accountDao",AccountDAO.class);  
   
 accountDAO.addAccount();  
   
 context.close();  
 }

1. Cream aspect. E bine sa cream un package si pentru ele

@Component  
@Aspect  
public class MyDemoLoggingAspect {  
  
 @Before("execution(public void addAcount())")  
 public void beforeAddAccountAdvice(){  
 System.*out*.println("\n=========>>>> Executing @Before advice on addAccount");  
 }  
}



Acum cand vom rula, vom vedea ca se va executa intai ce este in @Before **si** abea dupa metoda noastra din obiectul AccountDAO

***Aspect merge doar asupra beanurilor!!!***