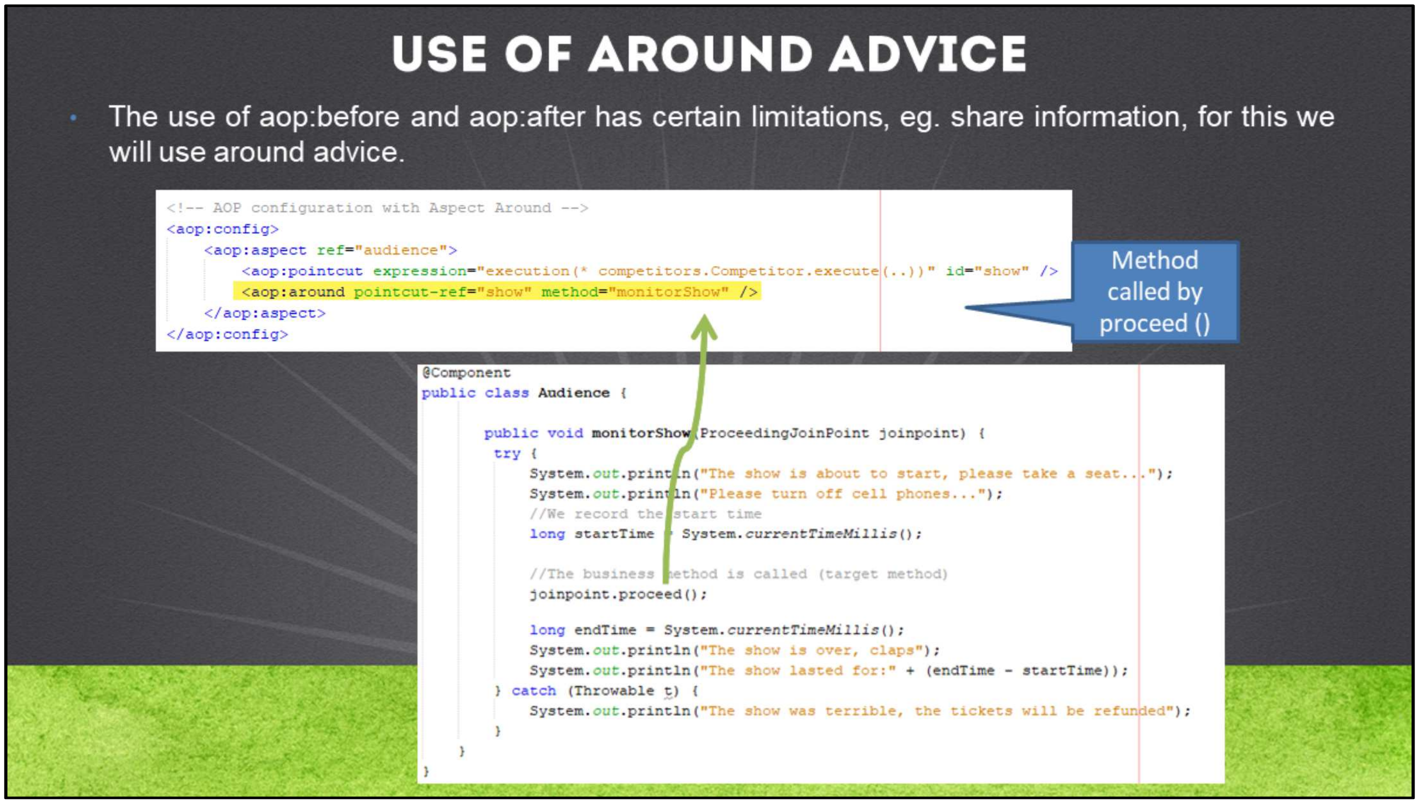
SPRING FRAMEWORK COURSE
ASPECT ORIENTED PROGRAMMING (PART 2)
By the expert: Eng. Ubaldo Acosta
Eng. Ubaldo Acosta
Global Mentoring
JAVA
UNIVERSITY
SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson ..

We will continue studying the topic of Aspect Oriented Programming (AOP) with Spring Framework.

Are you ready? Come on!



USE OF AROUND ADVICE

- The use of aop:before and aop:after has certain limitations, eg. share information, for this we will use around advice.

```xml
<!-- AOP configuration with Aspect Around -->
<aop:config>
    <aop:aspect ref="audience">
        <aop:pointcut expression="execution(* competitors.Competitor.execute(..))" id="show" />
        <aop:around pointcut-ref="show" method="monitorShow" />
    </aop:aspect>
</aop:config>
```

```java
@Component
public class Audience {

        public void monitorShow(ProceedingJoinPoint joinpoint) {
         try {
             System.out.println("The show is about to start, please take a seat...");
             System.out.println("Please turn off cell phones...");
             //We record the start time
             long startTime = System.currentTimeMillis();

             //The business method is called (target method)
             joinpoint.proceed();

             long endTime = System.currentTimeMillis();
             System.out.println("The show is over, claps");
             System.out.println("The show lasted for:" + (endTime - startTime));
         } catch (Throwable t) {
             System.out.println("The show was terrible, the tickets will be refunded");
         }
     }
}
```

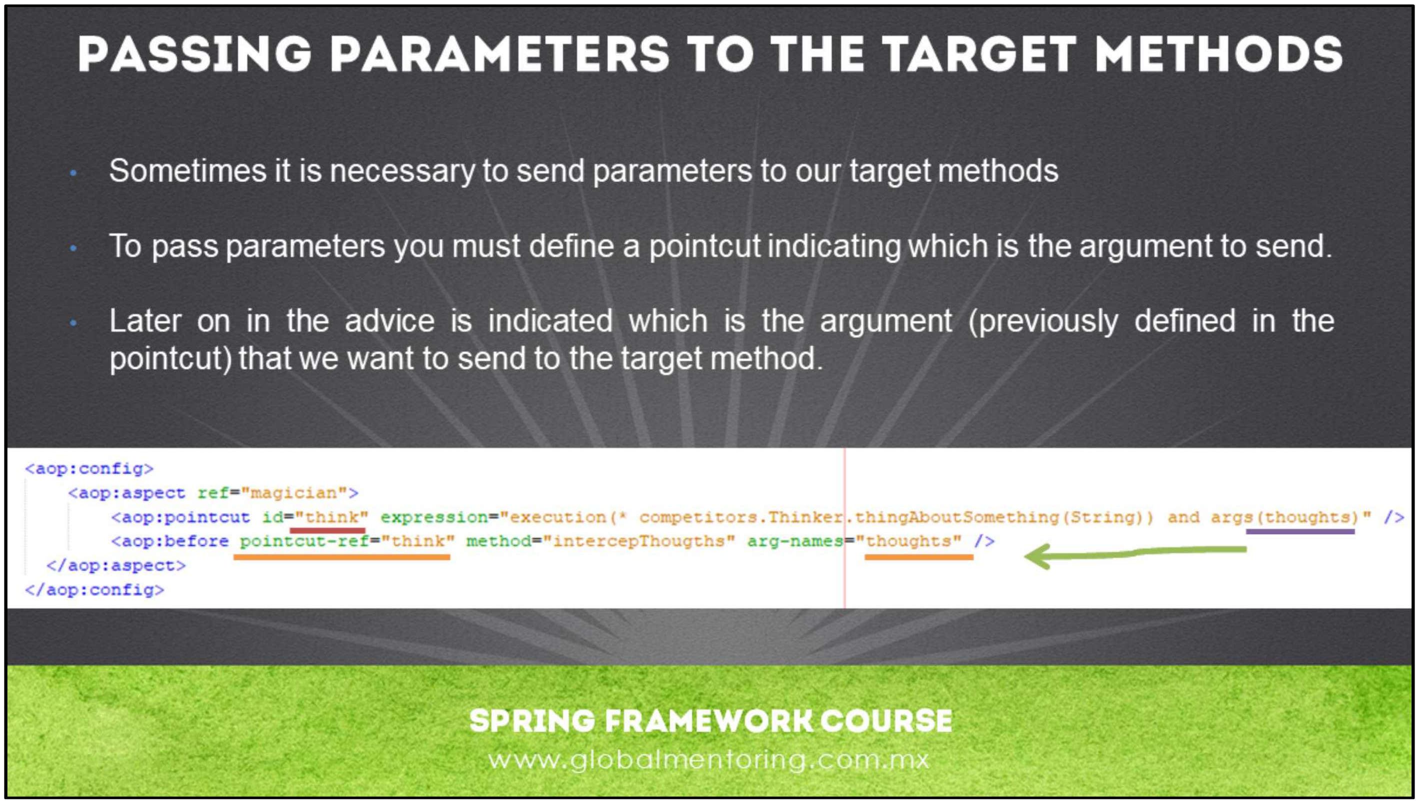
The use of aop:before and aop:after has certain limitations, for example sharing information before and after the execution of the business method.

To avoid these limitations we will use aop:around. To achieve this we will use the ProceedingJoinPoint object, which we receive as a parameter in our advice method.

The ProceedingJoinPoint object will allow us to execute the objective method (Competitor.execute) from our advice method (Audience.monitorShow). To delegate this call, the proceed() method will be used.

NOTE: If the call to the proceed() method is not included, it is as if we never finished the AOP flow, so we are responsible for making this call or the application will be blocked.

Another interesting question is that just as we can block access to the application, we can also use the proceed method to call the business method several times, this is interesting since it could be used as a retry logic if something failed with the business method, and we could even define how many attempts we will make to the business method.



PASSING PARAMETERS TO THE TARGET METHODS

- Sometimes it is necessary to send parameters to our target methods
- To pass parameters you must define a pointcut indicating which is the argument to send.
- Later on in the advice is indicated which is the argument (previously defined in the pointcut) that we want to send to the target method.

```xml
<aop:config>
    <aop:aspect ref="magician">
        <aop:pointcut id="think" expression="execution(* competitors.Thinker.thingAboutSomething(String)) and args(thoughts)" />
        <aop:before pointcut-ref="think" method="intercepThougths" arg-names="thoughts" />
    </aop:aspect>
</aop:config>
```

SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx

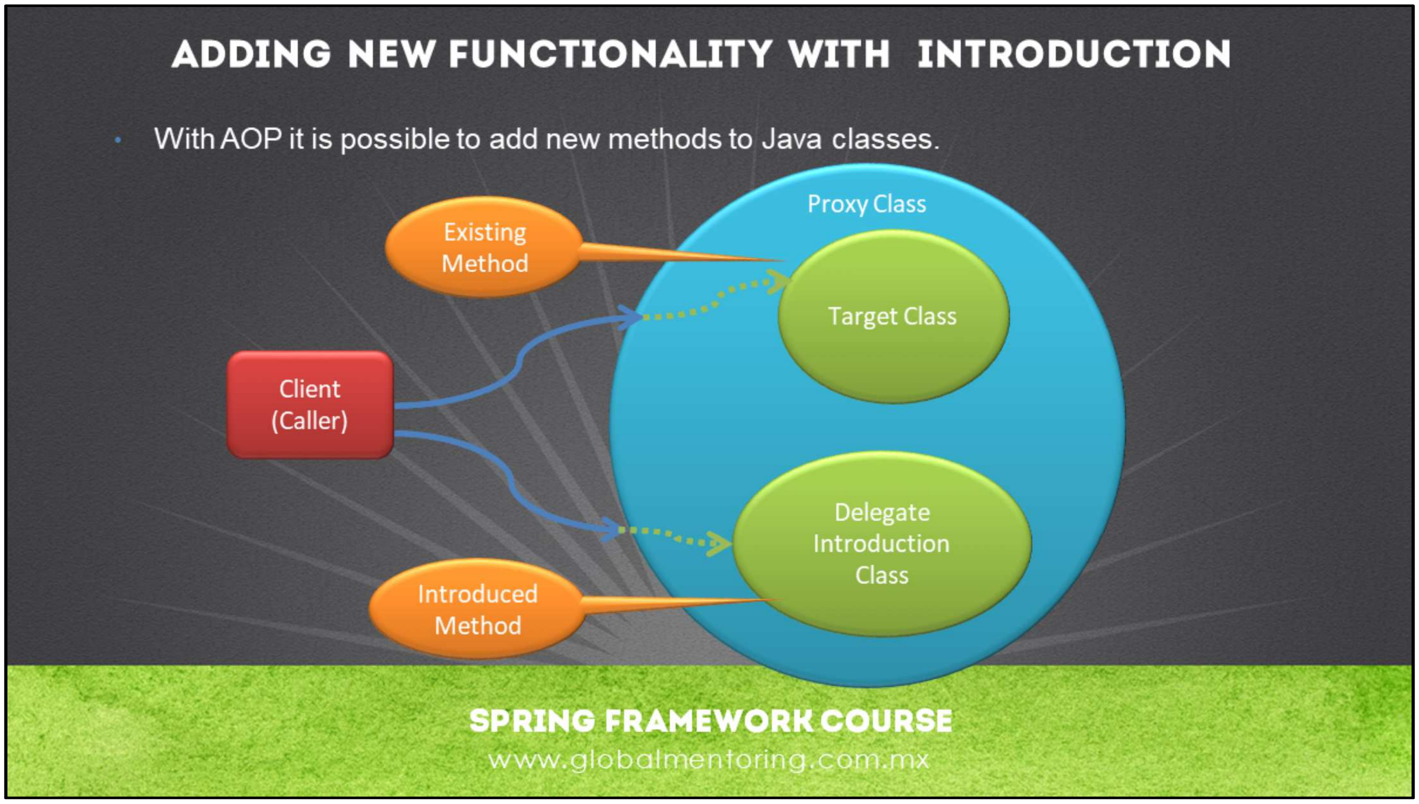
So far we have applied simple aspects, which do not require sharing information between the advice and the objective method.

However, if we need to share information between the notified methods and our notifiers, it is necessary to specify it explicitly.

As we can see in the figure, on the one hand, in the pointcut we specify that the method to be monitored (thinkAboutSomething) receives a parameter of type String, and we also specify the name of the argument of the method.

With this already configured, we can proceed to configure our advice, in which we add the attribute arg-names and indicate the name of the parameter that we want to send to our objective method called interceptThoughts().

We will later create an example to put this concept into practice.



ADDING NEW FUNCTIONALITY WITH INTRODUCTION

- With AOP it is possible to add new methods to Java classes.

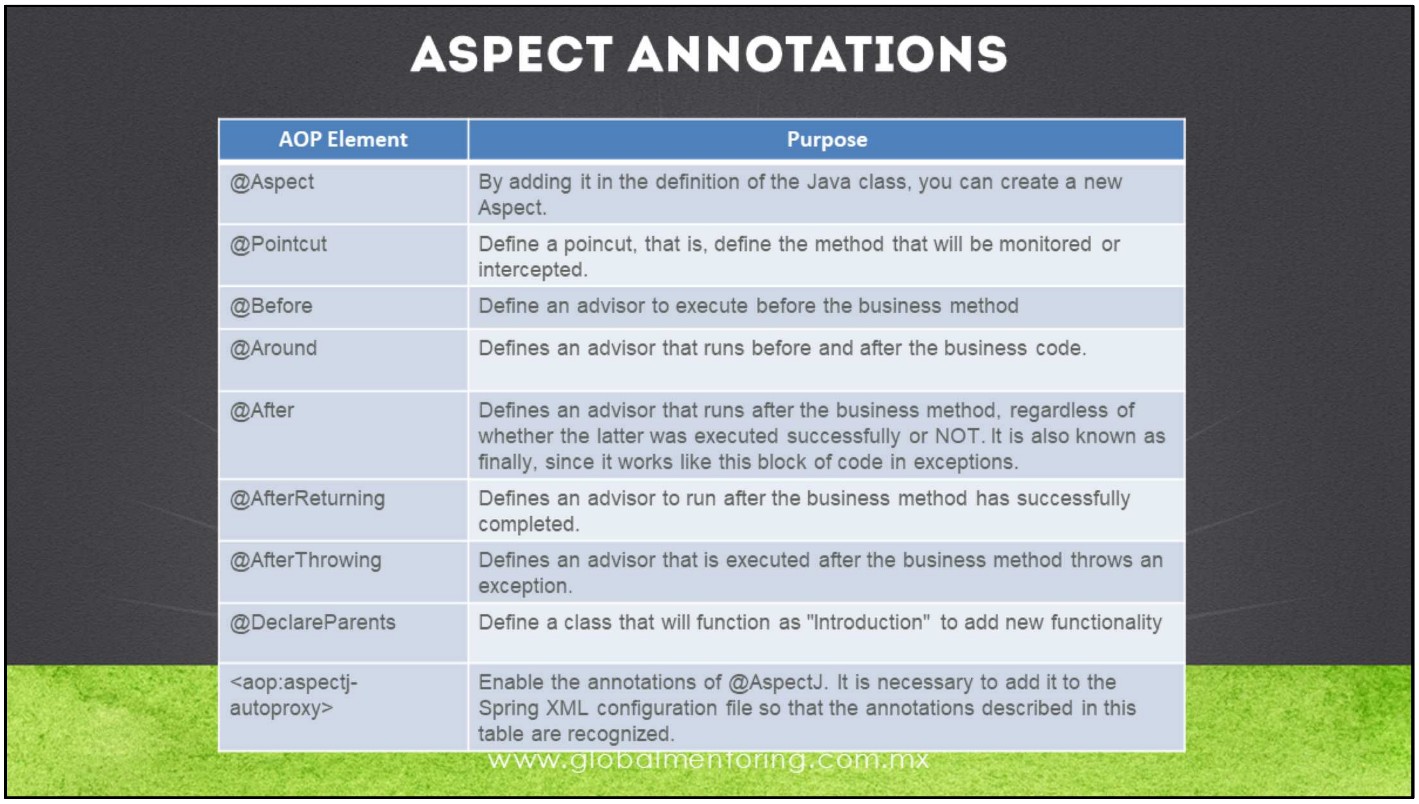
Some languages such as Ruby and Groovy have the concept of open classes, this means that we can add new methods to classes without changing the definition of the same.

In the case of Java this is not possible, however with the use of AOP and with the help of the concept known as Introduction it is possible to add new methods to existing classes, regardless of whether we have their source code or not.

```
<aop:config>
    <aop:aspect>
        <aop:declare-parents
              types-matching="competitors.Competitor+"
              implement-interface="competitors.Competitor"
              default-impl="competitors.InitialCandidate"/>
    </aop:aspect>
</aop:config>
```

In this example, we can see that we are declaring a new Parent type in the hierarchy of the Contestant class. The class Candidate can add new methods, and the default implementation is InitialCandidate.

With this we can allow the client class to call new methods to our target classes, intercepting the call and delegating it to a new class and new functionality. This makes the way to implement new functionality transparent even if we do not have the Java code of the Objective class, for example, when we work with third-party projects.



ASPECT ANNOTATIONS

AOP Element	Purpose
@Aspect	By adding it in the definition of the Java class, you can create a new Aspect.
@Pointcut	Define a poincut, that is, define the method that will be monitored or intercepted.
@Before	Define an advisor to execute before the business method
@Around	Defines an advisor that runs before and after the business code.
@After	Defines an advisor that runs after the business method, regardless of whether the latter was executed successfully or NOT. It is also known as finally, since it works like this block of code in exceptions.
@AfterReturning	Defines an advisor to run after the business method has successfully completed.
@AfterThrowing	Defines an advisor that is executed after the business method throws an exception.
@DeclareParents	Define a class that will function as "Introduction" to add new functionality
<aop:aspectj-autoproxy>	Enable the annotations of @AspectJ. It is necessary to add it to the Spring XML configuration file so that the annotations described in this table are recognized.

To activate the @AspectJ annotation handling, it is necessary to activate it in the Spring XML configuration file.

```
<aop:aspectj-autoproxy>
```

An example of the handling of Annotations is the following:

```
@Component
@Aspect
public class Audience {

    @Pointcut("execution(* competitors.Competitor.execute(..))")
    public void executeShow() {
    }

    @Before("executeShow()")
    public void sit() {
        System.out.println(“The show is about to begin, please take a seat...");
    }

    @Before("executeShow()")
    public void turnOffCells() {
        System.out.println(“Please turn off cell phones...");
    }

    @AfterReturning("executeShow()")
    public void clap() {
        System.out.println(“the show has ended, clap clap clap");
    }

    @AfterThrowing("executeShow")
    public void refund() {
        System.out.println(“The show was terrible, the tickets will be returned");
    }
}
```

Next, we will create an exercise to put this concept into practice.

ONLINE COURSE

SPRING FRAMEWORK

By: Eng. Ubaldo Acosta

SPRING FRAMEWORK COURSE

www.globalmentoring.com.mx