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

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

Are you ready? Come on!

One of the most important characteristics of Spring is the concept of Aspect Oriented Programming (AOP).

Let's imagine the following situation:

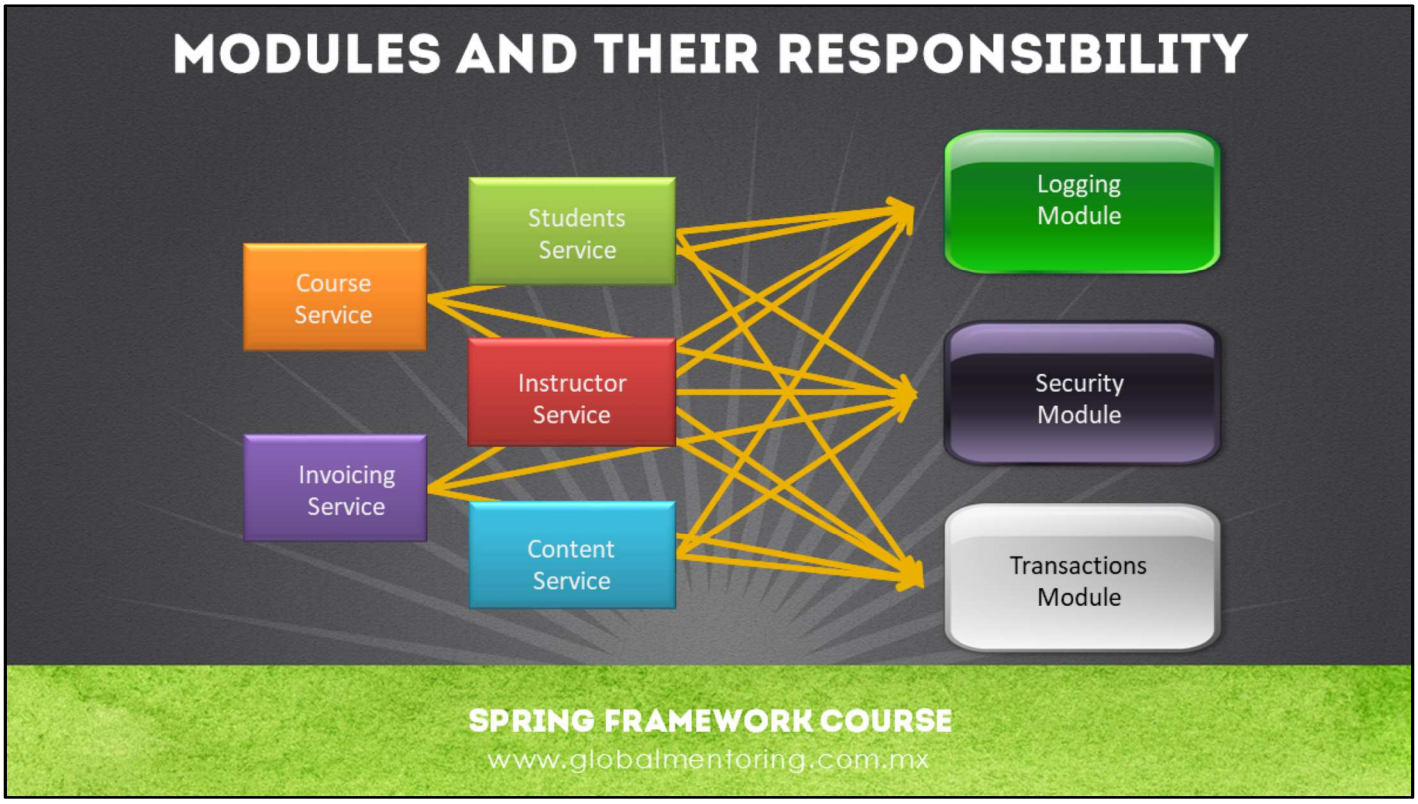
1. A man enters and / or leaves a building, whose doors are automatic.
2. The doors have sensors that allow them to open and close depending on the activity they detect.
3. Once the task is done, the door returns to its original state.
4. However, the sensors are always alert of the task they must perform.

Aspect Oriented Programming allows us to configure features that our modules or software components must perform. However, several of these characteristics are NOT the responsibility of the component to perform them directly.

AOP allows us to add functionality to one or several components in a non-intrusive way, allowing us to function as a sensor that is aware of the task that must be added / performed.

This allows us to add new functionality to our components or classes such as Transaction Management, Security, Logging, etc., without the need to add this code to the component itself, but delegate it to the classes that we will know as Aspects.

In turn these Aspects, are like sensors that are aware of the moment in which they should be executed (advise), and thus help and / or complement the tasks of our components in a non-intrusive way, that is, each one has its code by separately and in our case, Spring will be responsible for joining this functionality.



In software development, many of the activities are common to all applications. Transaction Management, Logging, Security, among others, are very important and necessary to include.

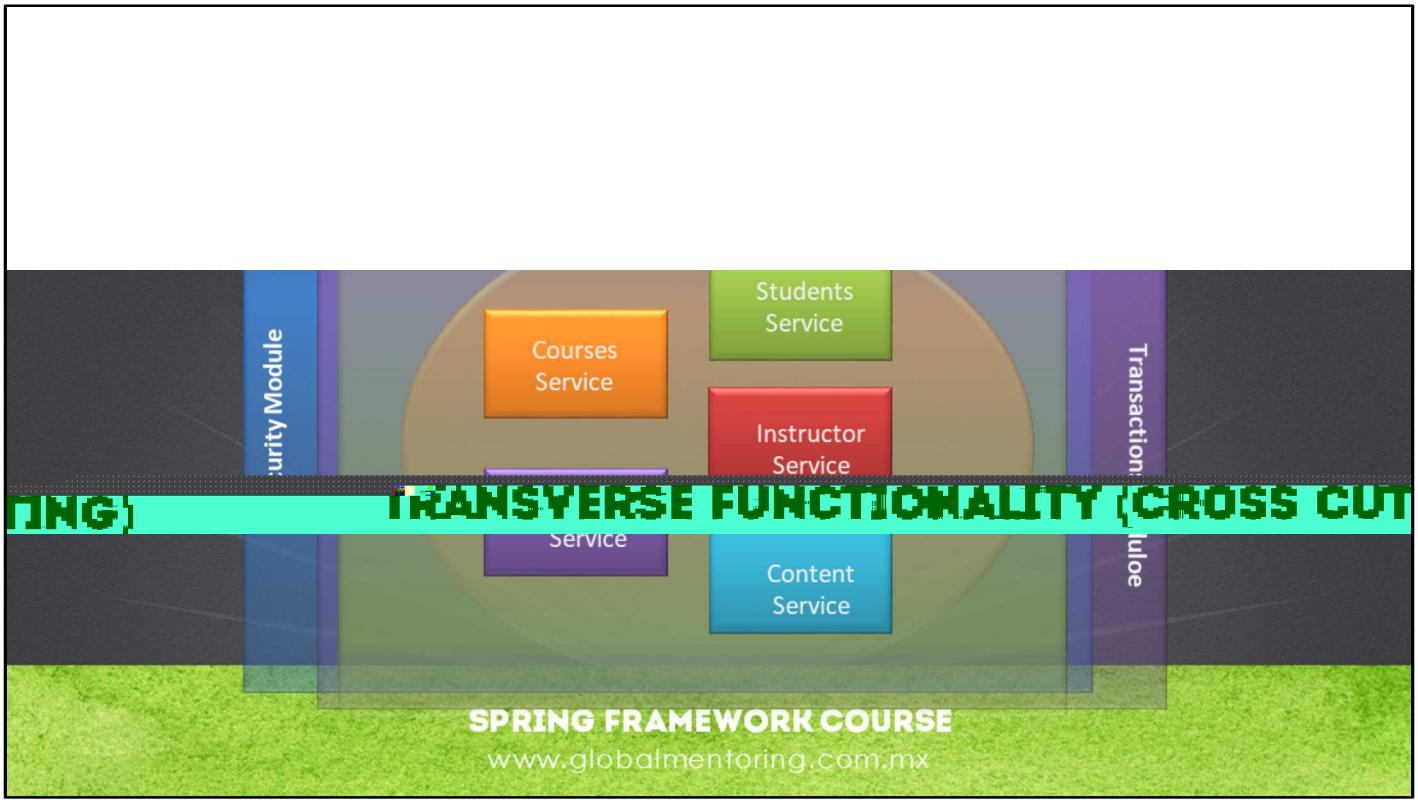
However, the question is: should our business components manage this functionality by themselves? Or should this functionality be activated automatically every time we need it?

The answer is obvious. Ideally, our components focus on managing the business tasks for which they were created, and all the functionality, like the aforementioned features, can be delegated to other components that we know as Aspects.

In software development, functionality that expands in a transactional manner or that is involved in more than one component is known as cross-cutting concerns.

In the figure we can see that many of our classes are directly involved with the Transversal Functionality, such as Logging, Security and Transaction Management.

The object-oriented programming would only allow to extend or inherit functionality of a single Transverse Aspect, but when having several, it is no longer possible to inherit this functionality.



AOP helps us to decouple the Transversal Functionality (Aspects), and thus avoid that our Service Modules are the ones that manage calls to the Aspects.

This allows our Business modules to focus on performing the tasks for which they were created.

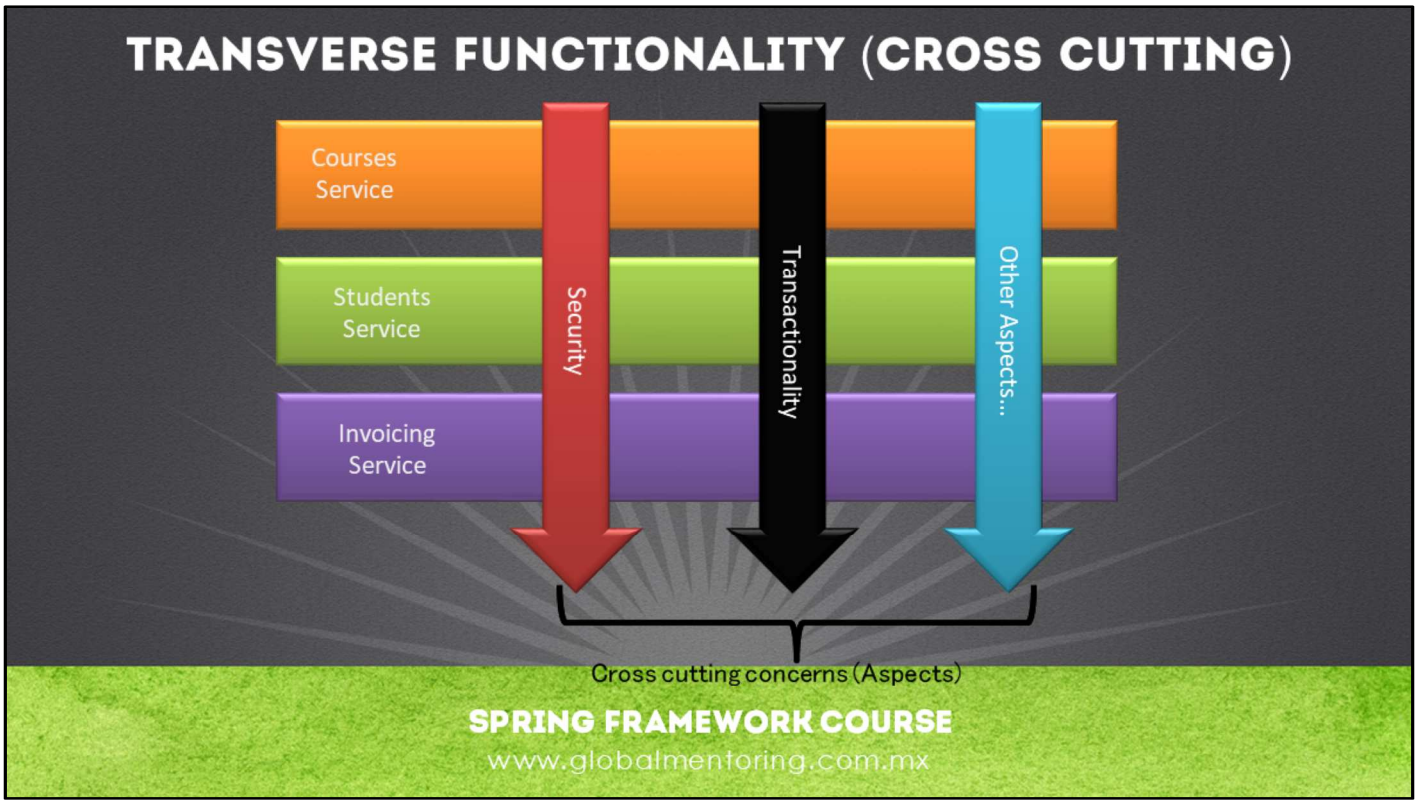
In this lesson we will learn how to configure our aspects using Spring, however the framework is not the only solution for programming aspects in Java.

AspectJ is one of the most popular frameworks, however its study is beyond the scope of this course.

For more information on the AspectJ project:

http://www.eclipse.org/aspectj

Several of the topics we will study is how Spring simplifies the way we will use AOP in our applications, including how to declare POJO classes to turn them into Aspects, as well as the use of annotations to create Aspects in Spring.



In the figure we can see another way of representing Cross-Functional Functionality (cruss-cotting), in which we see several of our business logic services, and which use the same functionality or aspects.

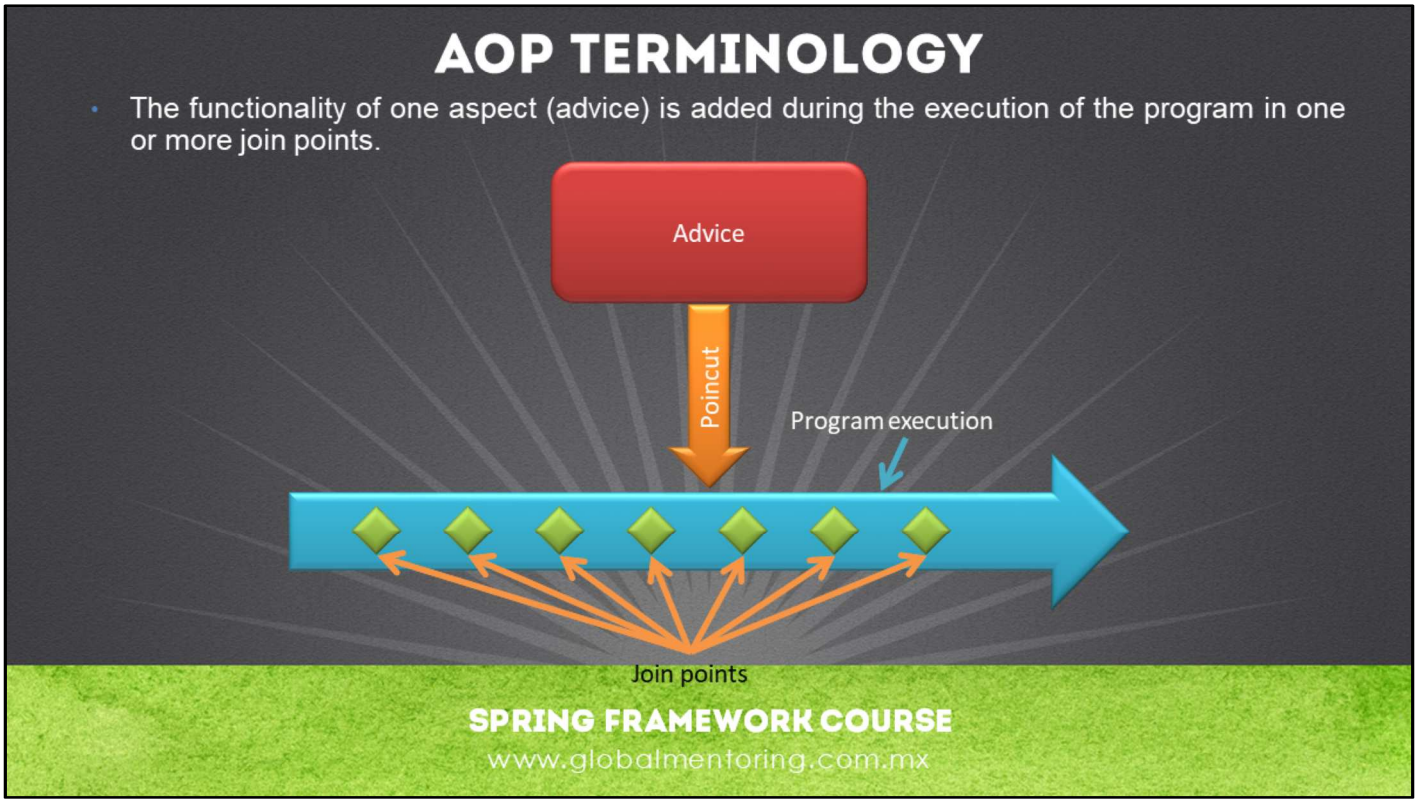
We can see that each of the Aspects is shared by one or more services. Being a common functionality, it is an ideal functionality to be programmed as an Aspect.

Spring allows you to convert POJOs into Java classes as robust as an EJB, without the need for a business container.

This has many advantages, from the point of view of lightness of our application, it is easy to test (testing), with low coupling and high cohesion between the modules.

Aspects offer an alternative to inheritance and code delegation. With AOP instead of adding lines of code to our business classes, we will simply use the Spring configuration file or better yet, annotations, to convert classes into Aspects. In a declarative way we can configure and relate our POJO classes and our aspects.

Business classes will not really know what aspects are associated, however, they can trust that, if they are well configured, these aspects or sensors will be activated at the right time and add the functionality that helps to complement the tasks of business of our modules. In our case, component or module, is any Java business class.



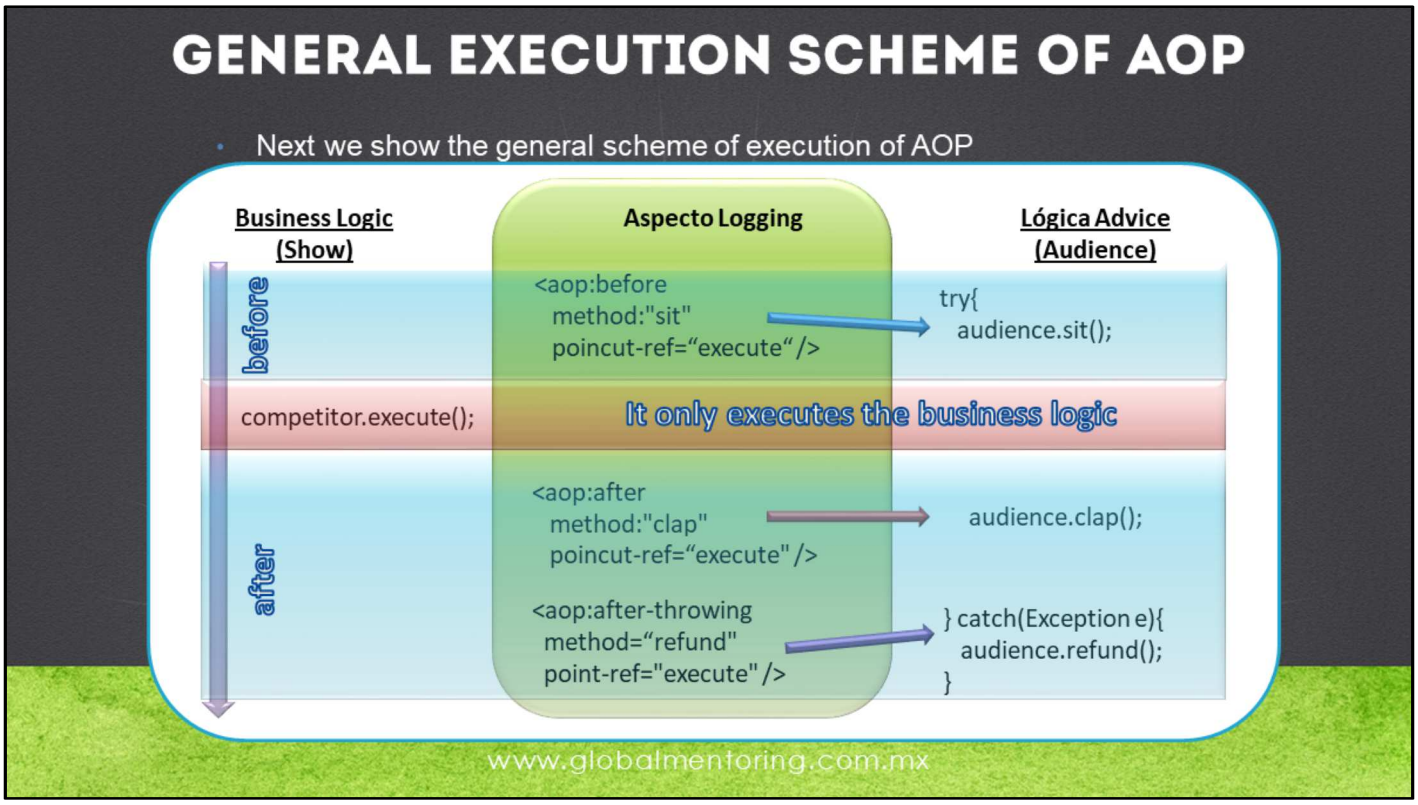
AOP has its own terms, which is very important to understand before you start coding. In the figure we can see several of the elements that come into play when applying AOP. The terms will be handled in English, as we will use them directly when coding.

Advice: It is the purpose of the Aspect itself, that is, the task to be executed. The advices define what task and when that aspect should be executed. For example, we can define whether an aspect should be defined before or after the call to the business method, since in some way the functionality of the aspect (s) "involves" the call to the business method.

Join Points: A Join point is the opportunity for advice to be applied / executed. It is a point in the execution of our program where advice can be applied, for example, a call to a method, an exception thrown or even the modification of the value of an attribute.

Pointcut: A pointcut allows to "point" to a join point, that is, they define where the advice should be applied. A pointcut can match one or more join points. Usually regular expressions are used to indicate which is the class and the method where the aspect will be applied.

Aspects: One aspect is the combination of advice and a pointcut. Once mixed, we obtain the what, when and where the aspect should be applied.



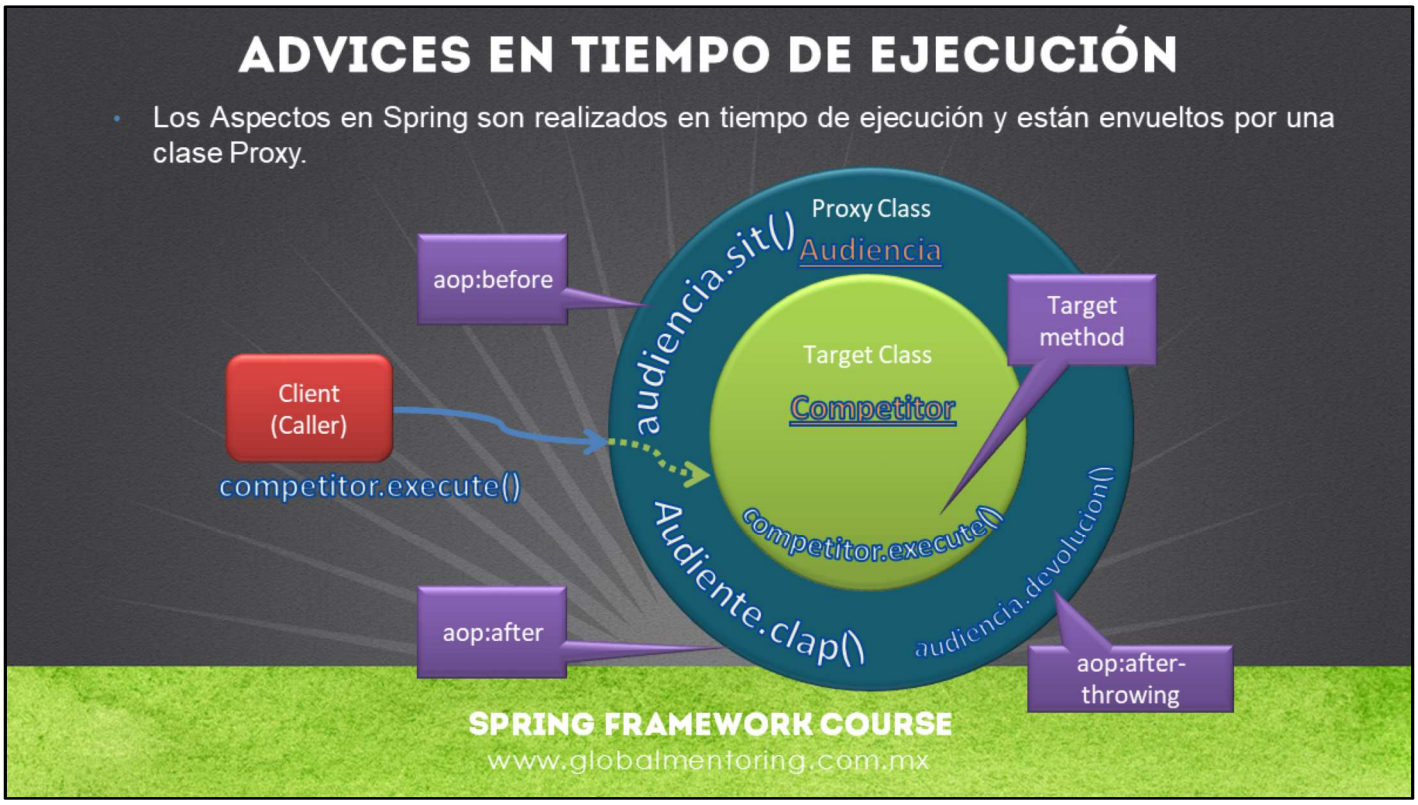
Continuing with more detail of the concepts of AOP, Spring defines 5 types of advices:

- Before: The functionality of the advice is executed before invocation of the business method.
- After: The functionality of the advice is executed after the business method.
- After-returning: The functionality of the advice is executed after the business method has been successfully completed.
- After-throwing: The functionality of the advice is executed after the method throws an exception.
- Around: It is the most used type, since it is executed before and after, involving the call of the business method.

In the figure we can see an example that before the show begins, the audience should take a seat and once the show is over, the audience should applaud, however if it was not to the public's liking, they can request the return of their tickets.

Then we can observe that when calling the competitor method, execute (), a logic is triggered that indicates that before and after the show it must perform a certain action. This is the basis of Aspect Oriented Programming.

From this moment we can see that this programming paradigm has many benefits. For example, the principle of transactional management is that in the "before" stage the transaction starts (aop: before), then the database query is executed, whether it is an insert, update or delete, this in the execution of the business logic, and finally makes a commit (aop: after) or a rollback (aop: after-throwing) if something failed during the execution of the query. We will study this topic in detail in later lessons.



Aspects in Spring are linked at runtime by wrapping them in a Proxy class.

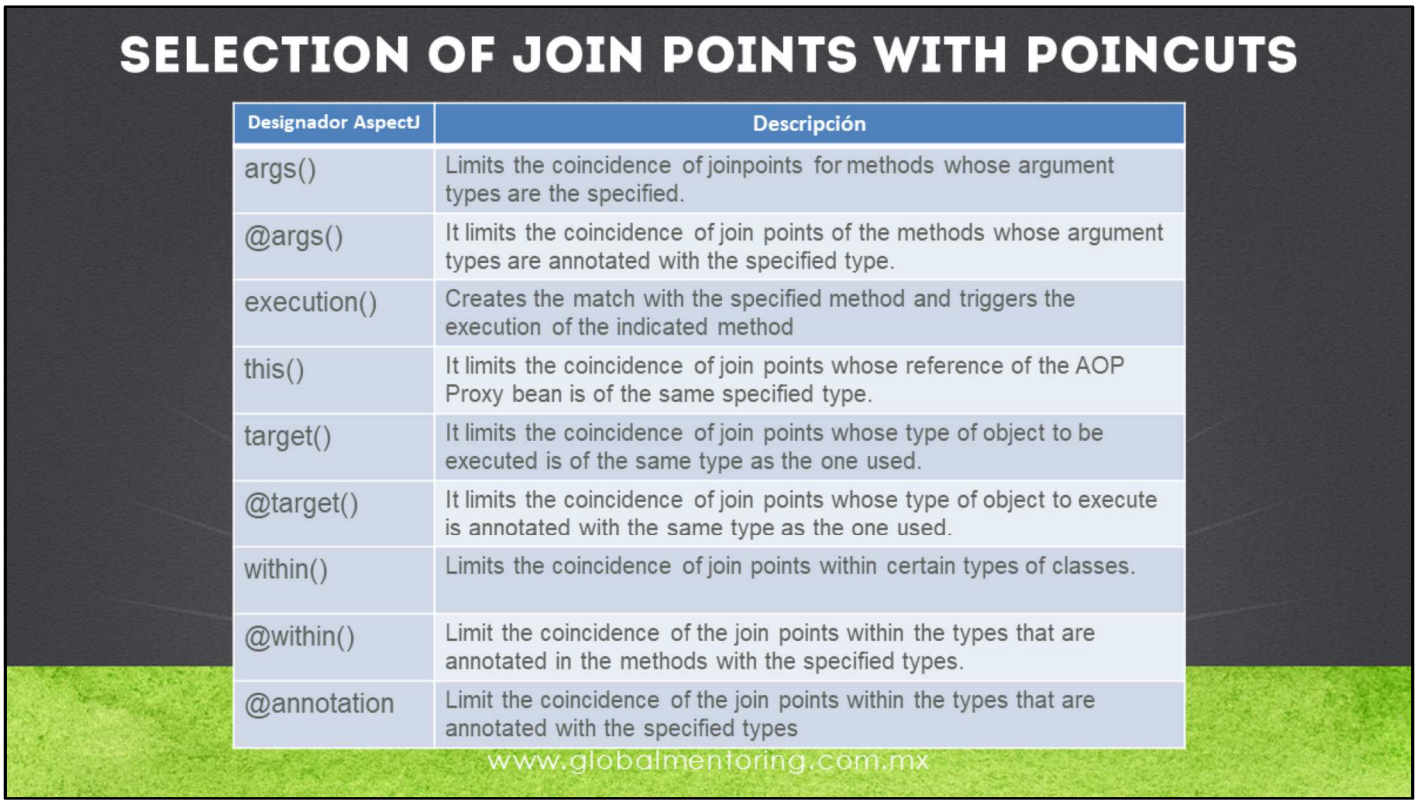
As we can see in the figure, the Proxy class has an objective bean (target bean). The task of the Proxy class is to be an intermediary between the Client Class (caller) and the target class (Target), by intercepting calls to the methods of the target class and wrapping it in the appearance functionality (which can be before and after from the call to the objective method) and later calls the method of the target class.

In short, the Proxy class is an interceptor of calls to a target class, and during this call the Proxy performs the logic of the Aspect, which can be before and / or after calling the method of the target class.

In the figure we can observe graphically the functioning of the proxy of the example of the previous sheet. In which, a client makes a call to the contestant method. Execute (), at this moment the Proxy class intercepts the call and performs the logic of the Aspect.

The aspect executes the functionality of aop: before, which in this case executes the audience.sit() method, later the proxy delegates the call to the business method, in this case it makes the call to the competitor.execute () method, which is really the call that interests the client, and finally the Proxy executes the call to the method aop: after called audience.clap(), and if any exception occurs the audience:refund() method is called configured with the aop: after-throwing .

We can see that everything is transparent both for the Client class and for the Competitor class, both are pure Java classes (POJO's), however Spring allows you to configure them very easily and convert them into Aspects (Audience class) and intercept the calls of the methods that we are interested (competitor.execute method.) From now on we begin to see the power of Spring and the flexibility it offers us to configure the requirements of our Java applications.



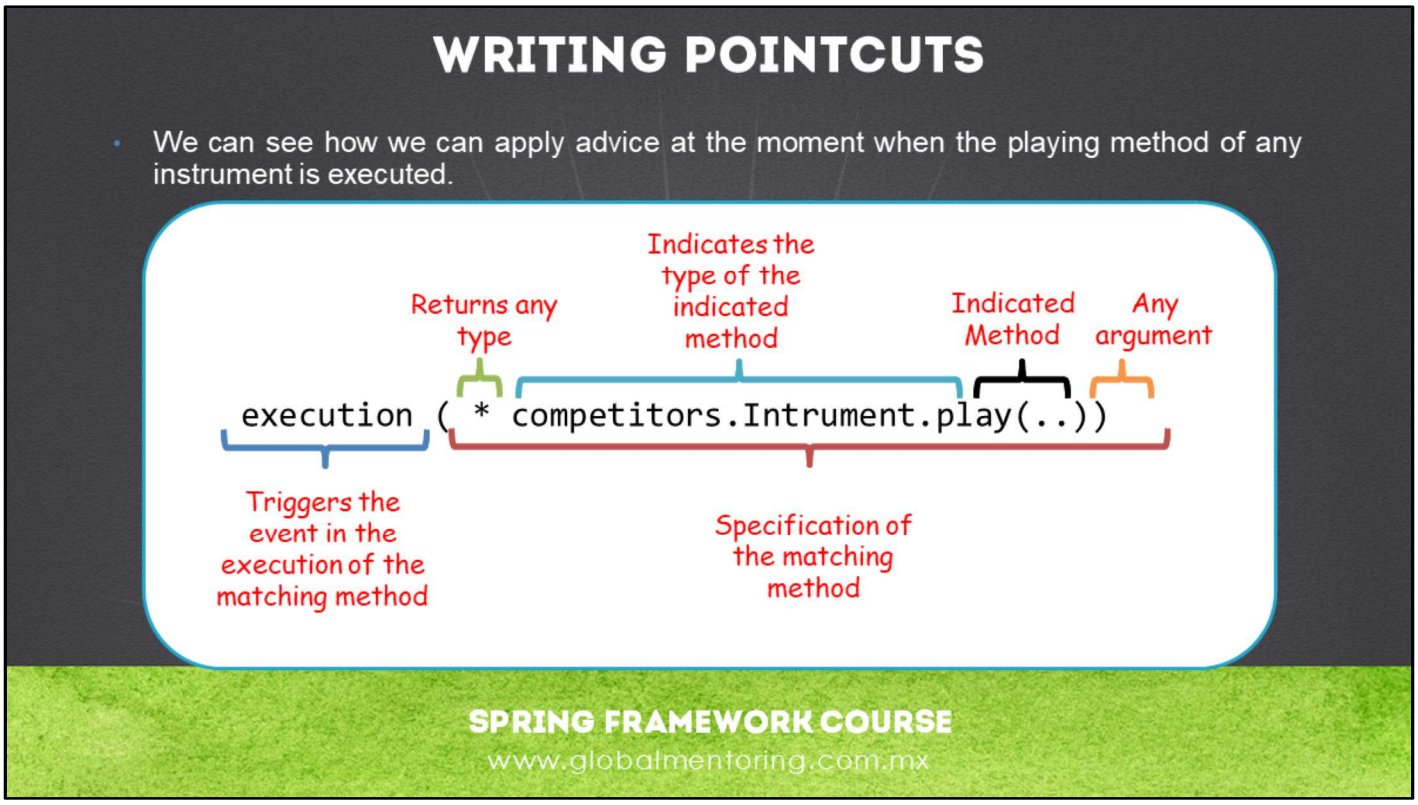
As we mentioned earlier, the pointcuts allow us to select where the advice will be executed. The pointcuts in Spring use the AspectJ Expression language.

The detailed study of AspectJ is beyond the scope of this course, however we will study what is necessary to feel comfortable creating pointcut expressions.

For a more detailed reference of the use of the Expression language and in general of AspectJ topics, the following pages can be consulted:

http://www.eclipse.org/aspectj/docs.php

http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html#some-example-pointcuts



In the figure we can show an example of how to write pointcuts.

The example shows how to use the execution designator to select the touch method of the Instrument type. In this case we indicate type and not class, since it can be a Java Interface or Class, and in the example we are using the most generic type, which is the Instrument Interface.

The specification of the matching method begins with the specification of the return type. When using * we indicate that it is any type of return.

Later we specify the type with its fully qualified name, that is, indicating the Java packages in which the type is found.

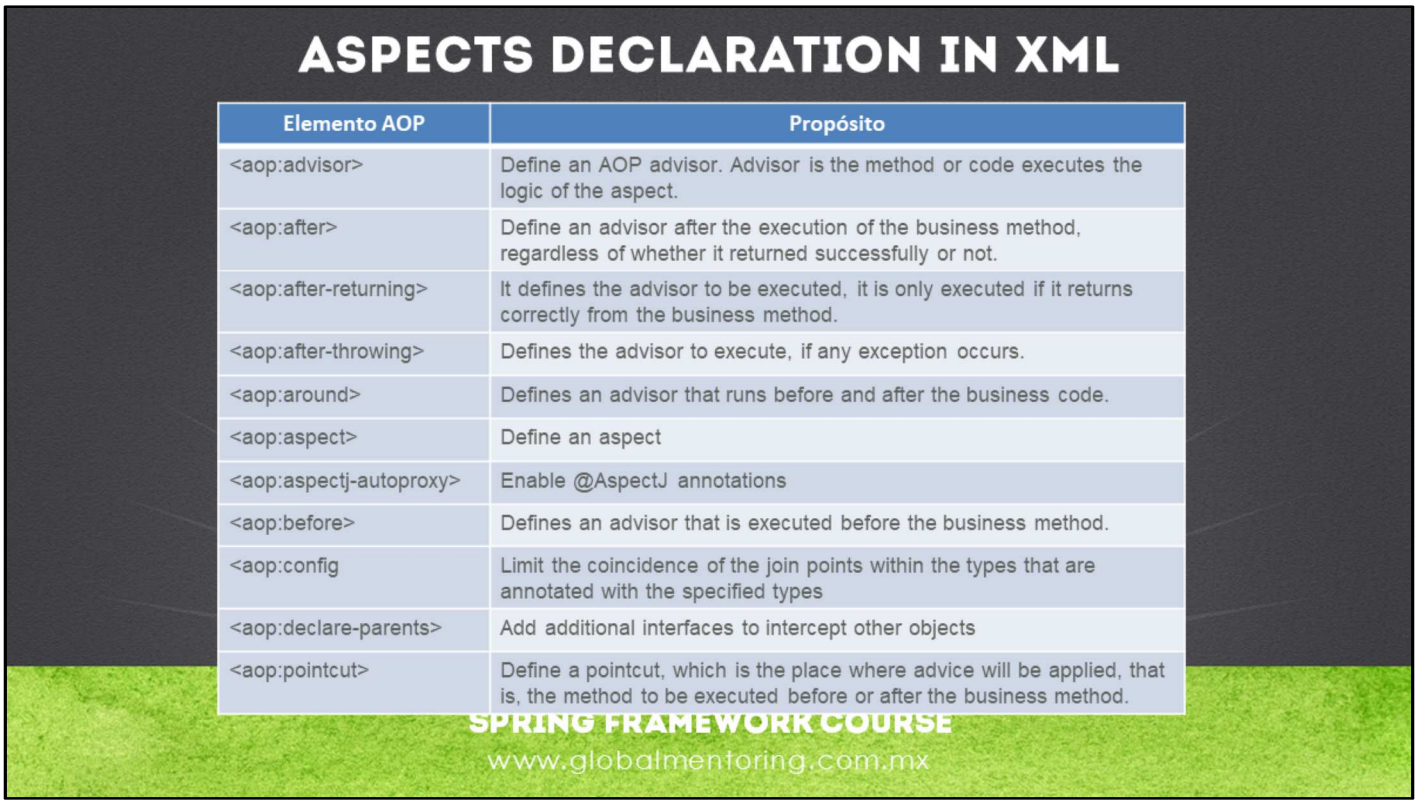
In turn we specify the name of the matching method, this is the most important, because when executing this method and in case of matching the indicated AspectJ expression, the aspect associated with this pointcut will be executed. This detail is not observed in this figure, since we will review it in later exercises.

Finally we specify the types of the arguments that we want to select in case of executing the indicated method. We will use the colon (..) to specify that the matching method can have any type and number of arguments, including No arguments.

Other examples are:

execution (* competitors.Instrument.play ()) && within (competitors. *)
Limit the scope of the pointcut to only what is inside the competitors package.

execution (* competitors.Instrument.play ()) && bean (pianist)
Limit the scope of the pointcut to just the beans called pianist



An example of the AOP configuration is as follows:

```
<!– AOP configuration -->
<aop:config>
    <aop:aspect ref="audience">

        <aop:pointcut expression="execution(* competitors.Competitor.execute(..))“ id="show" />

        <aop:before pointcut-ref="show" method="sit" />

        <aop:after-returning pointcut-ref="show" method=“clap" />
        <aop:after-throwing pointcut-ref="show" method=“refund"/>
    </aop:aspect>
</aop:config>
```

In this configuration we can see several of the elements already related.

We can see that we are creating an aspect called Audience, which will behave in a certain way before and after the show. Before the show they should sit down, and after the show they should applaud or ask for the return of their money, depending on whether the show was to their liking or not.

In the following exercise we will put the AOP concept into practice.