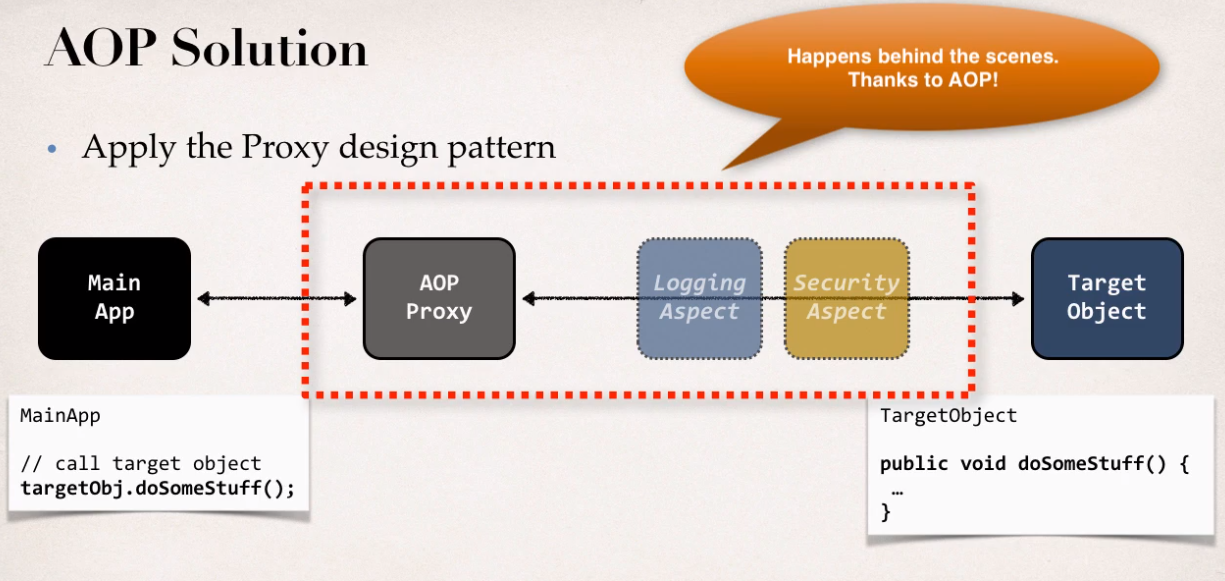
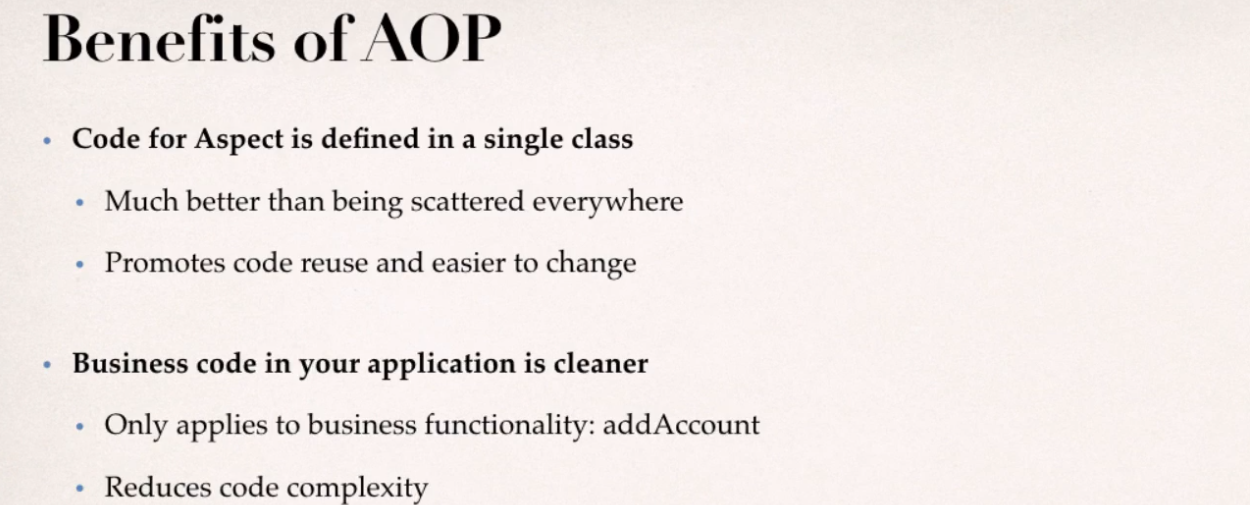
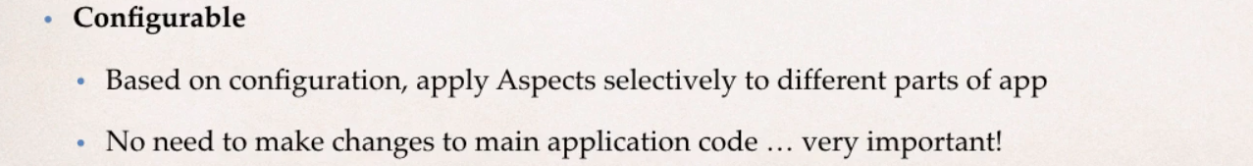
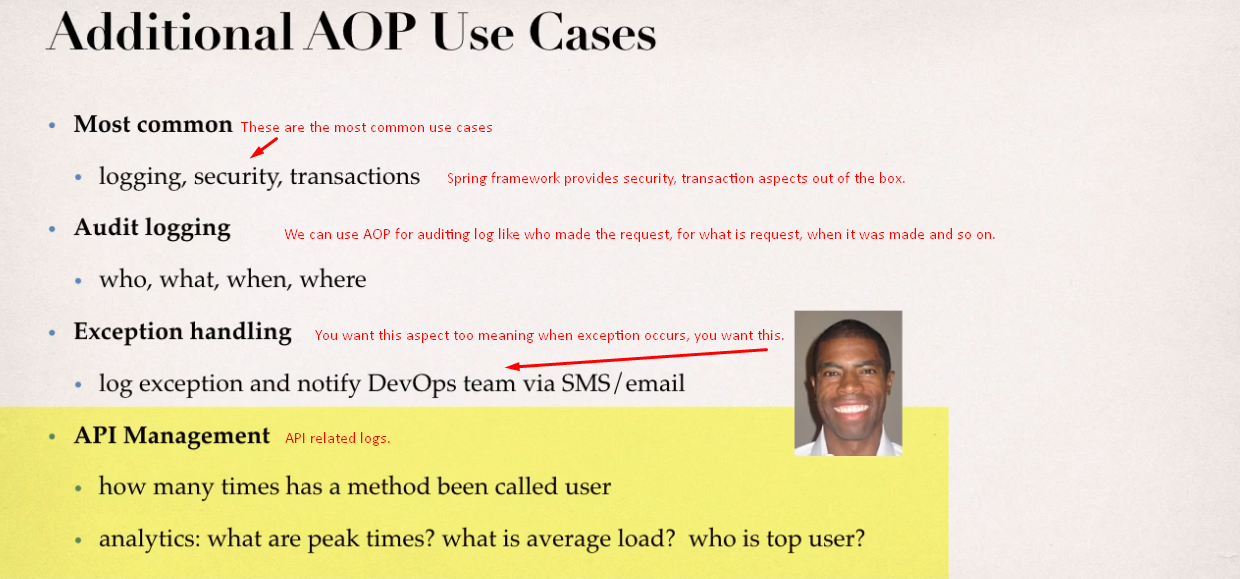
**One solution** to apply the AOP is **proxy design pattern.** You have main application which calls method on the **target object** say targetObject.doSomeStuff(). Your main application has no idea about proxy object. This is very similar to making a call to your best friend, but behind the scenes, you call is being monitored. So it’s being proxied, and you have these others that are listening on that line. So you have like logging aspect listening on your phone conversation, security aspect listening on your phone conversation and they can take action based on whatever you say, or you pass or whatever. These are spies in spy network. Main important is that this all happens behind the scene.   
So you write normal java code in main app and normal java code in target object & this all happens behind the scene. Thanks to AOP BUT you’ve to get it configured. We will see that later on in this videos.









If you have too many aspects, then your app flow can be hard to follow. So image a project with 20 developers or 100 developers, they’re each creating their own set of 10 to 20 aspects and applying them system wise, it’s ready hard to figure out what’s being called and so on. You can’t simply look at the main code. You’ve to go and look at other aspects and look at the configurations. So it can be so challenging. So the **key** here is that you use aspects in moderation and make some rules and governance on your team to have a good idea as far as the aspect development, as far as who’s creating them and who’s applying them.  
Also there is **minor performance cost** for aspect execution if you’re using **run-time weaving** which Spring AOP makes use of. It’s small performance hit not big deal. It’s like nanoseconds, milliseconds. But again, you want to overdo it.