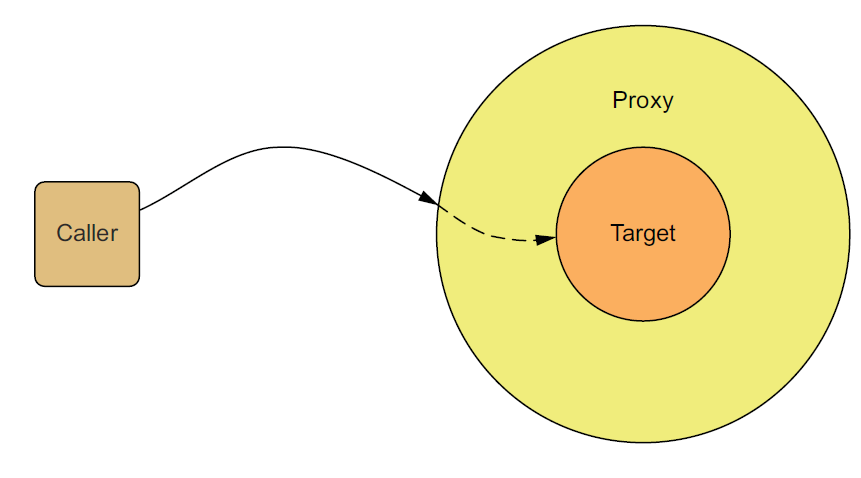
***Spring’s AOP support***

* Not all AOP frameworks are created equal. They may differ in how rich their join point models are. Some allow you to apply advice at the field-modification level, whereas others only expose the join points related to method invocations.
* They may also differ in how and when they weave the aspects. Whatever the case, the ability to create pointcuts that define the join points at which aspects should be woven is what makes it an AOP framework.
* We’ll focus on Spring AOP. Even so, there’s a lot of synergy between the Spring and AspectJ projects, and the AOP support in Spring borrows a lot from the AspectJ project.
* Spring’s support for AOP comes in four styles:
* Classic Spring proxy-based AOP
* Pure-POJO aspects
* @AspectJ annotation-driven aspects
* Injected AspectJ aspects (available in all versions of Spring)
* The first three styles are all variations on Spring’s own AOP implementation.
* Spring AOP is built around dynamic proxies.
* Consequently**, spring’s AOP support is limited to method interception.**
* When held up against simple declarative AOP and annotation-based AOP, Spring’s classic AOP seems bulky and overcomplicated. Therefore, *classic* Spring AOP won’t be covered .
* With Spring’s aop namespace, you can turn pure POJOs into aspects. In truth, those POJOs will only supply methods that are called in reaction to a pointcut. Unfortunately, this technique requires XML configuration, but it’s an easy way to declaratively turn any object into an aspect.
* Spring borrows AspectJ’s aspects to enable annotation-driven AOP. Under the covers, it’s still Spring’s proxy-based AOP, but the programming model is almost identical to writing full-blown AspectJ annotated aspects. The perk of this AOP style is that it can be done without any XML configuration.
* If your AOP needs exceed simple method interception (constructor or property interception, for example), you’ll want to consider implementing aspects in AspectJ. In that case, the fourth style listed will enable you to inject values into AspectJ-driven aspects.

**SPRING ADVICE IS WRITTEN IN JAVA**

* All the advice you create in Spring is written in a standard Java class. That way, you get the benefit of developing your aspects in the same integrated development environment (IDE) you’d use for normal Java development.
* The pointcuts that define where advice should be applied may be specified with annotations or configured in a Spring XML configuration, but either will be familiar to Java developers.
* Contrast this with AspectJ. Although AspectJ now supports annotation-based aspects, it also comes as a language extension to Java. This approach has benefits and drawbacks. By having an AOP-specific language, you get more power and fine-grained control, as well as a richer AOP toolset. But you’re required to learn a new tool and syntax to accomplish this.

****

**SPRING ADVISES OBJECTS AT RUNTIME**

* In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. As illustrated.
* The proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean.
* Between the time when the proxy intercepts the method call and the time when it invokes the target bean’s method, the proxy performs the aspect logic.
* Spring doesn’t create a proxied object until that proxied bean is needed by the application.
* If you’re using an ApplicationContext, the proxied objects will be created when it loads all the beans from the BeanFactory.
* Because Spring creates proxies at runtime, you don’t need a special compiler to weave aspects in Spring’s AOP.

**SPRING ONLY SUPPORTS METHOD JOIN POINTS**

* As mentioned earlier, multiple join-point models are available through various AOP implementations.
* Because it’s based on dynamic proxies, Spring only supports method join points.
* This is in contrast to some other AOP frameworks, such as AspectJ and JBoss, which provide field and constructor join points in addition to method pointcuts.
* Spring’s lack of field pointcuts prevents you from creating very fine-grained advice, such as intercepting updates to an object’s field.
* And without constructor pointcuts, there’s no way to apply advice when a bean is instantiated
* But method interception should suit most, if not all, of your needs. If you find yourself in need of more than method interception, you’ll want to complement Spring AOP with AspectJ.