Spring AOP

Let say I am logging about the business logic but want to separate it from the main business logic in separate file but somehow we need to connect them.

This type of concern is known as cross-cutting concerns.

Terms in aspect oriented programming:

Aspect: the class which contains the concern which needs to be integrated with the main logic.

Joint point: It's a point during the execution of the program. Let say out of many methods in main logic I want to maintain log for certain bunch of methods. We need to connect them with example such as cross cutting concern. So these are join points. In spring, joint point always points towards method execution.

Advice : it's just the handled method for the joint point. Like we make log for business function saveEntity method of controller.

pointCut: wildcard used to match the joint point. So it basically connect the advice with joint point by specifying where applicable.

Target object: The target object is the object being advised by one or more aspects. It is the object on which the advised methods are invoked. The target object usually implements one or more interfaces or extends one or more classes.

AOP proxy: an object created by aop framework in order to implement the aspect contracts (advice method execution and so on). In spring framework an AOP proxy will be a JDK dynamic proxy or CGLIB proxy.

introduction: declaring additional methods or fields on behalf of a type. Spring AOP allows one to introduce new interfaces to any advised object. For example we can make a bean implement an isModified interface using introduction, to simply caching.

Weaving: linking aspects with other applications types or objects to create advised object. Spring Aop performs at runtime but can be done at compile, or load time also. This forms the actual connection of actual method with advice.

In Spring AOP, the advised object and the target object can refer to the same object, but they serve slightly different purposes:

1. **Target Object**: The target object is the original object on which a proxy is created. It's the object that clients interact with directly, and it's the object that

contains the business logic that we want to intercept with advice.

2. **Advised Object**: The advised object is the target object that has been wrapped with a proxy to apply cross-cutting concerns, such as logging, security, or transaction management. This proxy intercepts method calls to the target object and invokes the appropriate advice before or after the method execution.

So, while they can be conceptually the same, in Spring AOP, when we refer to the "target object," we typically mean the original object before any advice is applied. The "advised object" refers to the target object after it has been wrapped with a proxy and advice has been applied to it.

There are five types of advices:

- Before advice:
- After returning advice - @AfterReturning
- After throwing advice: if method exists by throwing an error - @AfterThrowing
- After (finally) advice: advice to be existed in normal or when an exception is thrown. - @After
- Around advice:

For normal spring add spring aop dependency.

Packages concerned are org.aspectj.lang.*

So to make an aspect got to class and add @Aspect like say on Logging class.

Do not forget to make it a component.

To make a method inside the logging class as joint point:

```
@Before(“execution(public List<Alien>
com.telusko.springmvcboot.AlienController.getAliens())”)
public void log() {
}
```
We have to give full qualified name.
We are executing some method.
This will not work as for Alien also we have to give full qualified name.
We can remove that.

```
@Before(“execution(public *
com.telusko.springmvcboot.AlienController.getAliens())”)
public void log() {
}
```

To get the logger class (we will use SLF4J class)
```
private static final Logger logger =
LoggerFactory.getLogger(AspectLogger.class);
```

We can specify the type of logging in spring boot using application.properties:

```
logging.level.root=info
```
We can also save the log in other file:

```
logging.app=app.log
```

We can find the file under target folder.

Access Information about the joint point

To access information about the joint point in an advice method in Spring AOP, you can use the `JoinPoint` parameter. The `JoinPoint` interface provides methods to retrieve various details about the intercepted method and its execution context.

Here's an example of how you can use `JoinPoint` in an advice method:

```java
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class MyAspect {

    @Before("execution(* com.example.MyService.*(..))")
    public void beforeMethod(JoinPoint joinPoint) {
        // Get the intercepted method signature
        String methodName = joinPoint.getSignature().getName();

        // Get the intercepted method arguments
        Object[] args = joinPoint.getArgs();

        // Print out information about the intercepted method
        System.out.println("Intercepted method: " + methodName);
        System.out.println("Arguments: " + Arrays.toString(args));
    }
}
```

In this example:

- The `beforeMethod()` advice method takes a `JoinPoint` parameter.
- Inside the method, `joinPoint.getSignature()` returns the signature of the intercepted method, which includes its name, return type, and declaring type.
- `joinPoint.getArgs()` returns an array of objects representing the arguments passed to the intercepted method.

You can use other methods of the `JoinPoint` interface to access more information about the joint point, such as the target object (`joinPoint.getTarget()`), the kind of advice being executed (`joinPoint.getKind()`), etc.

Yes, the `JoinPoint` interface in Spring AOP provides several other methods in addition to `getSignature()` and `getArgs()`. Some of the commonly used methods include:

- `getTarget()`: Returns the target object being advised.
- `getThis()`: Returns the proxy object that the advice is being invoked on.
- `getKind()`: Returns the kind of advice (e.g., Before, After, Around).
- `getSourceLocation()`: Returns the source location of the advised method.
- `toLongString()`: Returns a long string representation of the joint point, including details like method modifiers, return type, method name, and parameter types.

These methods allow you to access various details about the intercepted method and its execution context, providing flexibility in implementing advice logic based on specific requirements.

In Spring AOP, when the advice is applied to a method of a bean, the target object and the proxy object can indeed be the same. This is because Spring AOP typically uses proxy-based mechanism to apply aspects to beans.

- **Target Object**: This refers to the actual object being advised. It's the object that the aspect is intended to interact with when the advised method is invoked.

- **Proxy Object**: Spring creates a proxy object around the target object to intercept method calls and apply the advice. This proxy object implements the same interface(s) or extends the same class(es) as the target object, allowing it to substitute the target object seamlessly.

However, in more complex scenarios, such as when using AspectJ aspects or certain configurations with Spring AOP, the target object and proxy object may differ, especially in cases involving method calls within the same object or interactions across different aspects. In such cases, the `getTarget()` and `getThis()` methods of the `JoinPoint` interface will return different objects.

It's essential to understand the behavior of target and proxy objects in your specific Spring AOP configuration to ensure correct implementation of advice logic.