educación IT
CODE YOUR FUTURE

Bootcamp Java Developer

Fase 3 - Java Architect

Módulo 29

POA + Spring

POA en Spring

Spring posee un muy buen módulo de POA y sus desarrolladores han considerado que no era necesaria una sintaxis propia ya que existe la de AspectJ - ampliamente probada en la práctica-.

La implementación completa de AspectJ permite usar también el acceso a campos, la llamada a un constructor, y otros, aunque esto en POA de Spring no es posible.

Se debe tener presente que no es lo mismo usar la sintaxis de AspectJ que usar AspectJ en sí.

Anotaciones vs. XML

Hay dos **sintaxis alternativas** para usar POA en Spring:

1. Mediante el **uso de anotaciones** en el propio código Java.
2. Con **etiquetas en un archivo** de configuración.

Sin embargo, mediante el uso de anotaciones podemos encapsular el POA junto con el código Java en un único lugar. En teoría, este es el sitio en que debería estar si el POA es un requisito de negocio que debe cumplir la clase.

Para **añadir soporte POA a un proyecto Spring** serán necesarias **dos librerías**: *aspectjweaver.jar*, y *aspectjrt.jar*.

Además, si se quiere **utilizar POA con clases que no implementen ninguna interfaz**, se necesitaría la librería *CGLIB*.

Puntos de corte (pointcuts)

Recordemos: un *pointcut* es un punto de interés en el código antes, después o alrededor del cual queremos ejecutar algo (un *advice*).

- Un *pointcut* no puede ser cualquier línea arbitraria de código.
- La versión actual de Spring solo soporta puntos de corte en ejecuciones de métodos de *beans*.

Es importante destacar que al definir un *pointcut* realmente no estamos todavía diciendo que vayamos a ejecutar nada, simplemente marcamos un "punto de interés". **La combinación de *pointcut + advice* es la que realmente hace algo útil.**

Expresiones más comunes

La expresión más usada en *pointcuts* de Spring es **`execution()`**, que representa la **llamada a un método** que encaje con una determinada firma. Se puede especificar la firma completa del método incluyendo tipo de acceso (`public`, `protected`, y otros), tipo de retorno, nombre de clase (incluyendo paquetes), nombre de método y argumentos.

Tener en cuenta:

1. El tipo de acceso y el nombre de clase son opcionales, pero no así el resto de elementos.
2. Se puede usar el comodín * para sustituir a cualquiera de ellos, y también el comodín, que sustituye a varios tokens, por ejemplo varios argumentos de un método, o varios subpaquetes con el mismo prefijo.

3. En los parámetros:
   - **()** indica un método sin parámetros,
   - **(..)** indica cualquier número de parámetros de cualquier tipo, y podemos también especificar los tipos, por ejemplo: (`String`, `*`, `int`) indicaría un método cuyo primer parámetro es `String`, el tercero `int` y el segundo puede ser cualquiera.

Por ejemplo, para especificar todos los métodos con acceso "`public`" de cualquier clase dentro del paquete *ar.com.spring.aop* indicaríamos:

```
execution( public * ar.com.spring.aop.*.*(..))
```

Se pueden combinar *pointcuts* con los operadores lógicos **&&**, **||** y **!**, con el mismo significado que en el lenguaje C, por ejemplo:

```
// todos los getters o setters de cualquier clase

execution( public * get*()) || execution( public void set*(*))
```

El operador **&&** se suele usar en conjunción con `args` como una forma de "dar nombre" a los parámetros, por ejemplo:

```
execution( public void set*(*)) && args(nuevoValor)
```

Pointcuts con nombre

Se puede asignar un nombre arbitrario a un *pointcut* (lo que se denomina una *firma*). Esto permite referenciarlo y reutilizarlo de manera más corta y sencilla que si tuviéramos que poner la expresión completa que lo define.

La definición completa consta de la anotación **`@Pointcut`** seguida de la expresión que lo define y la signatura. Para definir la firma se usa la misma sintaxis que para definir la de un método Java en un interfaz. Eso sí, el valor de retorno debe ser `void`.

Por ejemplo:

```
@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}

@Pointcut("execution(public * get*())")
public void unGetterCualquiera() {}

@Pointcut("within(es.ua.jtech.ejemplo.negocio.*"))
public void enNegocio() {}

@Pointcut("unGetterCualquiera() && enNegocio()")
public void getterDeNegocio() {}
```

Advices

Los *advices* son la pieza del puzzle que faltaba para que todo cobre sentido. **Un *advice* es algo que hay que hacer en cierto punto de corte**, ya sea antes, después, o "alrededor" (antes y después) del punto.

Se especifican con una anotación con el *pointcut* y la definición del método Java a ejecutar (firma y código). Como en Spring, **los puntos de corte deben ser ejecuciones de métodos.**

Los casos posibles son:

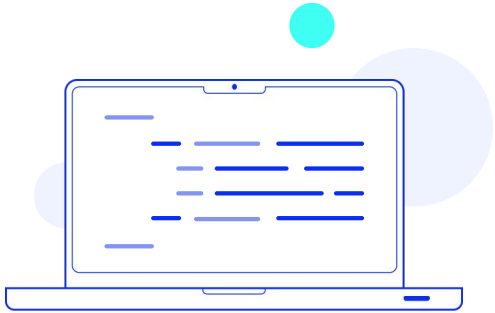
1. Antes de la ejecución de un método (anotación **`@Before`**).
2. Después de la ejecución normal, es decir, si no se genera una excepción (anotación **`@AfterReturning`**).
3. Después de la ejecución con excepción/es (anotación **`@AfterThrowing`**).
4. Después de la ejecución, se hayan producido o no excepciones (anotación **`@After`**).
5. Antes y después de la ejecución (anotación **`@Around`**).

Un aspecto *(aspect)* es un conjunto de *advices*.

Con la sintaxis de AspectJ, los aspectos se representan como clases Java, marcadas con la anotación **`@Aspect`**.

En Spring, además, un aspecto debe ser un *bean*, por lo que tendremos que anotarlo como tal:

```
import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

@Component
@Aspect
public class EjemploDeAspecto{
	//aquí vendrían los advices...
}
```



@Before

Esta anotación **ejecuta un *advice* antes de la ejecución del punto de corte** especificado. Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EjemploBefore {
	@Before("execution(public * get*())")
	public void controlaPermisos() {
		// ...
	}
}
```

Ejecutaría `controlaPermisos()` antes de llamar a cualquier `getter`.

@AfterReturning

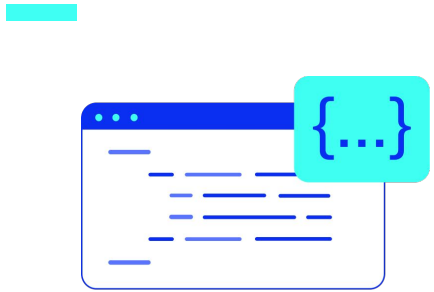
Esta anotación **ejecuta un *advice* después de la ejecución del *pointcut*** especificado, siempre que el método del punto de corte retorne de forma normal (sin generar excepciones). Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class EjemploAfterReturning {
	@AfterReturning("execution(public * get*())")
	public void log() {
		// ...
	}
}
```

Para hacer `log`, nos puede interesar saber el valor retornado por el método del punto de corte. Este valor es accesible con la sintaxis alternativa:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class EjemploAfterReturning {
       @AfterReturning(
          pointcut="execution(public * get*())",
          returning="valor")
          public void log(Object valor) {
          // ...
          }
       }
}
```

Al poner `Object` como tipo de la variable asociada al valor de retorno, estamos indicando que nos da igual el tipo que sea (incluso si es primitivo). Especificando un tipo distinto, podemos reducir el ámbito del *advice* para que solo se aplique a los puntos de corte que devuelvan un valor del tipo deseado.



@After

Esta anotación ejecuta un *advice* después de la ejecución del punto de corte especificado, genere o no una excepción, es decir, al estilo del `finally` de Java.

@Around

Esta anotación ejecuta parte del *advice* antes y parte después de la ejecución del punto de corte especificado.

La filosofía consiste en que el usuario es el que debe especificar en el código del *advice* en qué momento se debe llamar al punto de corte. Por ello, el *advice* debe tener como mínimo un parámetro de la clase **`ProceedingJoinPoint`**, que representa el punto de corte. Al llamar al método **`proceed()`** de esta clase, ejecutamos el punto de corte.

Por ejemplo:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class EjemploAround {
      @Around("execution(public * get*())")
      public Object ejemploAround(ProceedingJoinPoint
pjp) throws Throwable {
            System.out.println("ANTES");
            Object valorRetorno = pjp.proceed();
            System.out.println("DESPUES");
            return valorRetorno;
        }
      }
```

¡Sigamos trabajando!