

Spring Core : la programmation orientée aspect

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



spring

- 1 Introduction
- 2 Terminologies
- 3 AOP avec AspectJ
 - Avec les annotations
 - Avec le XML

Programmation Orientée Aspect

- proposée par la société **Xerox** en 1996
- un nouveau paradigme de programmation
- une nouvelle façon de structurer le code d'une application
- ~~un langage de programmation~~
- applicable sur les langages de POO (**C++**, **Java...**) ou sur les langages procéduraux (**C...**)

Spring

Programmation Orientée Aspect

- proposée par la société **Xerox** en 1996
- un nouveau paradigme de programmation
- une nouvelle façon de structurer le code d'une application
- ~~un langage de programmation~~
- applicable sur les langages de POO (**C++**, **Java**...) ou sur les langages procéduraux (**C**...)

Objectif

Améliorer les langages de programmation (et ne pas les remplacer)

Programmation Orientée Objet

- permet de regrouper des données et des traitements selon la sémantique dans des classes
- facilite la maintenance, la réutilisation et l'extension

© Achrel L.

Programmation Orientée Objet

- permet de regrouper des données et des traitements selon la sémantique dans des classes
- facilite la maintenance, la réutilisation et l'extension

Limite de la POO : exemple

La gestion des traces (Logging) \Rightarrow Répétition du code

Explication

- Si nous voulons afficher un message avant et/ou après et/ou pendant l'exécution de chaque méthode de l'application
- Nous devons ajouter un `System.out.print(...)` ; dans chaque méthode

© Achret L

Spring

Explication

- Si nous voulons afficher un message avant et/ou après et/ou pendant l'exécution de chaque méthode de l'application
- Nous devons ajouter un `System.out.print(...)` ; dans chaque méthode

Constats

- Trop répétitif
- Trop long

Avec la programmation orientée aspect

- Nous pouvons définir un aspect qui capture les appels de méthodes dont le code
- Implémenter ou utiliser un `Logger` qui affiche la trace.

© Achref EL M...

Spring

Avec la programmation orientée aspect

- Nous pouvons définir un aspect qui capture les appels de méthodes dont le code
- Implémenter ou utiliser un `Logger` qui affiche la trace.

Constats

- Rapide
- Facile à mettre en place
- Facile à supprimer aussi

Quelques cas d'utilisation de l'**AOP**

- Gestion de transaction
- Logging
- Gestion de la sécurité

Classe vs Aspect

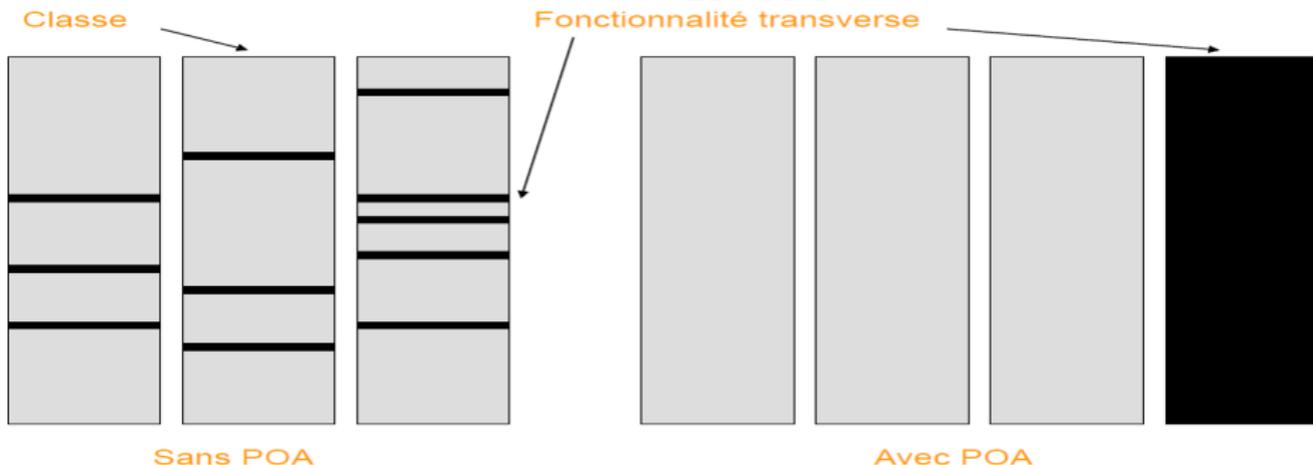
- Une classe = un motif, un plan, un moule, une usine...
- Un aspect = une fonctionnalité (de traces, de sécurités, de persistance des données)

© Achref EL MOU...

Spring

Classe vs Aspect

- Une classe = un motif, un plan, un moule, une usine...
- Un aspect = une fonctionnalité (de traces, de sécurités, de persistance des données)



Spring

Question

À quel moment faut-il appeler l'aspect ?

© Achref EL MOUELHI ©

Spring

Question

À quel moment faut-il appeler l'aspect ?

Réponse

jamais

Spring

Question

À quel moment faut-il appeler l'aspect ?

Réponse

jamais

Question : Alors comment faire ?

- Il nous faut un tisseur (weaver) : le système responsable de l'insertion
- son rôle : greffer l'ensemble des aspects sur l'ensemble des classes du programme

Quelques tisseurs d'aspects pour Java

- AspectJ : extension orientée aspect, créée à Xerox PARC, pour le langage de programmation Java (open-source) et utilisé par **Spring Framework**.
- JbossAOP : extension orientée aspect, créée par JBoss, pour le langage de programmation Java.
- JAC (Java Aspect Components) : extension orientée aspect français pour le langage de programmation Java (open-source).
- AspectWerkz : extension orientée aspect fusionnée avec AspectJ.
- ...

Spring

Tissage (Weaving)

Opération automatique consistant à insérer des aspects dans le programme initial.

© Achref EL MOUETRI

Spring

Tissage (Weaving)

Opération automatique consistant à insérer des aspects dans le programme initial.

Aspect

Un programme (une classe) contenant un greffon et un ou plusieurs points de coupes.

Spring

Tissage (Weaving)

Opération automatique consistant à insérer des aspects dans le programme initial.

Aspect

Un programme (une classe) contenant un greffon et un ou plusieurs points de coupes.

Greffon ? point de coupe ?

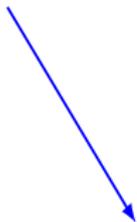
```
public void méthode() {  
    ...  
}
```

© Achref EL MOUELHI ©

```
@Before("execution(public void saluer(..))")  
public void méthode() {  
    ...  
}
```

Spring

Greffon



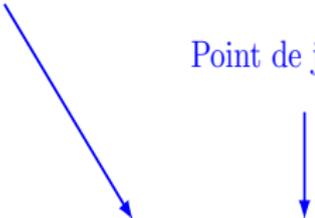
```
@Before("execution(public void saluer(..))")  
public void méthode() {  
    ...  
}
```

© Achref EL MOUJELHI ©

Spring

Greffon

Point de jonction



```
@Before("execution(public void saluer(..))")  
public void méthode() {  
    ...  
}
```

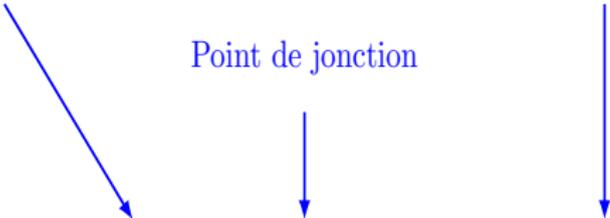
© Achref EL MOUJELHI ©

Spring

Greffon

Expression de coupe

Point de jonction



```
@Before("execution(public void saluer(..))")  
public void méthode() {  
    ...  
}
```

Spring

Greffon

Expression de coupe

Point de jonction

```
@Before("execution(public void saluer(..))")  
public void méthode() {  
    ...  
}
```

Code de greffon

Spring

Greffon (Advice)

Un bloc de code à insérer et exécuter.

© Achref EL MOUELHI ©

Spring

Greffon (Advice)

Un bloc de code à insérer et exécuter.

Point de jonction (JoinPoint)

Un endroit de l'application autour duquel un ou plusieurs aspects pourront être connectés.

Spring

Greffon (Advice)

Un bloc de code à insérer et exécuter.

Point de jonction (JoinPoint)

Un endroit de l'application autour duquel un ou plusieurs aspects pourront être connectés.

Point de coupe (PointCut)

Un ensemble de point de jonction.

AOP avec **AspectJ** : deux solutions possibles

- Avec les annotations (depuis la version 2.5 de **Spring Framework**)
- Avec le **XML**

Cinq types de greffons

- `before` : avant l'exécution de méthodes
- `around` : autour de l'exécution de méthodes (avant et après)
- `after throwing` : si une exception est levée
- `after returning` : après une exécution normale de méthodes
- `after` : après l'exécution de méthodes et quelle que soit la sortie (qu'une exception soit levée ou non)

Spring

Pour inclure AspectJ dans notre projet, on ajoute les dépendances suivantes dans la section `dependencies` de `pom.xml`

```
<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.9.6</version>
  <scope>runtime</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjrt -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.9.6</version>
</dependency>
```

Spring

Exemple : considérons l'interface `European` et les deux classes `French` et `English` utilisées dans le chapitre précédent

```
public interface European {  
    public void saluer();  
}
```

La classe `French`

```
@Component  
public class French implements European{  
    public void saluer() {  
        System.out.println("Bonjour");  
    }  
}
```

La classe `English`

```
@Component  
public class English implements European{  
    public void saluer() {  
        System.out.println("Hello");  
    }  
}
```

Spring

La classe `Main` et le fichier de configuration `applicationContext.xml` (en cas de XML Config)

```
public class App {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");  
        European e = (European) context.getBean("french");  
        e.saluer();  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:context="http://www.springframework.org/schema/context"  
    xmlns:p="http://www.springframework.org/schema/p"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  
        http://www.springframework.org/schema/context  
        http://www.springframework.org/schema/context/spring-context-4.0.xsd">  
  
    <context:component-scan base-package="org.eclipse.nation" ></context:component-scan>  
  
</beans>
```

Spring

La classe `Main` et le fichier de configuration `ApplicationConfig` (en cas de Java Config)

```
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(ApplicationConfig.
            class);
        European e = (European) context.getBean("french");
        e.saluer();
    }
}
```

```
package org.eclipse.configuration;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("org.eclipse.nation")
public class ApplicationConfig {

}
```

Spring

Créons une classe `TestAspect` **dans** `org.eclipse.aop`

```
package org.eclipse.aop;  
  
public class TestAspect {  
  
}
```

© Achref EL MOULALI

Spring

Créons une classe `TestAspect` **dans** `org.eclipse.aop`

```
package org.eclipse.aop;  
  
public class TestAspect {  
  
}
```

Pour que `TestAspect` **soit un Aspect, on ajoute l'annotation** `@Aspect`

```
package org.eclipse.aop;  
  
import org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
public class TestAspect {  
  
}
```

Spring

Autres annotations à ajouter

- Pour que l'aspect soit intégré dans le programme, on ajoute l'annotation `@Component`
- Pour que **Spring** active la configuration des aspects, on ajoute l'annotation `@EnableAspectJAutoProxy`

Spring

Nouveau contenu du `TestAspect`

```
package org.eclipse.aop;

import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.
    EnableAspectJAutoProxy;

@EnableAspectJAutoProxy
@Component
@Aspect
public class TestAspect {

}
```

**Il faut aussi scanner le package des aspects dans
applicationContext.xml (en cas de XML Config)**

```
<context:component-scan  
    base-package="org.eclipse.nation, org.eclipse.aop" >  
</context:component-scan>
```

© Achref EL

Spring

**Il faut aussi scanner le package des aspects dans
applicationContext.xml (en cas de XML Config)**

```
<context:component-scan  
    base-package="org.eclipse.nation, org.eclipse.aop" >  
</context:component-scan>
```

Ou dans ApplicationConfig (en cas de Java Config)

```
@ComponentScan("org.eclipse.nation, org.eclipse.aop")
```

Spring

Hypothèse

on voudrait afficher un message

- avant
- et après

l'exécution de la méthode `saluer`

Nouveau contenu de TestAspect

```
public class TestAspect {  
  
    @Before("execution(public void org.eclipse.nation.French.saluer(..)")  
    public void avant() {  
        System.out.println("Avant saluer");  
    }  
  
    @After("execution(public void org.eclipse.nation.French.saluer(..)")  
    public void apres() {  
        System.out.println("Après saluer");  
    }  
}
```

© Achref EL MOU

Nouveau contenu de `TestAspect`

```
public class TestAspect {  
  
    @Before("execution(public void org.eclipse.nation.French.saluer(..)")  
    public void avant() {  
        System.out.println("Avant saluer");  
    }  
  
    @After("execution(public void org.eclipse.nation.French.saluer(..)")  
    public void apres() {  
        System.out.println("Après saluer");  
    }  
}
```

Explication

- `@Before` : déclaration d'un greffon
- `execution` : type de point de jonction
- `System.out.println("Avant saluer");` : code du greffon
- `public void org.eclipse.nation.French.saluer(..);` : expression de coupe

Spring

En exécutant, le résultat est :

```
Avant saluer  
Bonjour  
Après saluer
```

Spring

On peut aussi utiliser les `wildcards` pour les expressions de coupe : sur le type de retour

```
public * org.eclipse.nation.French.saluer(..)
```

© Achref EL MOUELHI ©

Spring

On peut aussi utiliser les `wildcards` pour les expressions de coupe : sur le type de retour

```
public * org.eclipse.nation.French.saluer(..)
```

Sur le nom de la classe ou l'interface

```
public * org.eclipse.nation.*nch.saluer(..)
```

© Achref EL MOUADJIB

Spring

On peut aussi utiliser les `wildcards` pour les expressions de coupe : sur le type de retour

```
public * org.eclipse.nation.French.saluer(..)
```

Sur le nom de la classe ou l'interface

```
public * org.eclipse.nation.*nch.saluer(..)
```

Sur le nom de la méthode

```
public * org.eclipse.nation.*nch.*(..)
```

Spring

On peut aussi utiliser les `wildcards` pour les expressions de coupe : sur le type de retour

```
public * org.eclipse.nation.French.saluer(..)
```

Sur le nom de la classe ou l'interface

```
public * org.eclipse.nation.*nch.saluer(..)
```

Sur le nom de la méthode

```
public * org.eclipse.nation.*nch.*(..)
```

Sur le nom du package

```
public * *.*nch.*(..)
```

Spring

On peut aussi utiliser les `wildcards` pour les expressions de coupe : sur le type de retour

```
public * org.eclipse.nation.French.saluer(..)
```

Sur le nom de la classe ou l'interface

```
public * org.eclipse.nation.*nch.saluer(..)
```

Sur le nom de la méthode

```
public * org.eclipse.nation.*nch.*(..)
```

Sur le nom du package

```
public * *.*nch.*(..)
```

On peut aussi omettre la visibilité (par défaut `public`)

```
* *.*nch.*(..)
```

Spring

On peut aussi filtrer selon le nombre et/ou le type des paramètres : 0 ou plusieurs paramètres

```
public * org.eclipse.nation.French.saluer(..)
```

© Achref EL MOUELHI ©

Spring

On peut aussi filtrer selon le nombre et/ou le type des paramètres : 0 ou plusieurs paramètres

```
public * org.eclipse.nation.French.saluer(..)
```

Sans paramètres

```
public * org.eclipse.nation.French.saluer()
```

© Achref EL MOUËL

Spring

On peut aussi filtrer selon le nombre et/ou le type des paramètres : 0 ou plusieurs paramètres

```
public * org.eclipse.nation.French.saluer(..)
```

Sans paramètres

```
public * org.eclipse.nation.French.saluer()
```

Selon le type des paramètres

```
public * org.eclipse.nation.French.saluer(org.eclipse.model.Personne,  
String)
```

Spring

On peut aussi filtrer selon le nombre et/ou le type des paramètres : 0 ou plusieurs paramètres

```
public * org.eclipse.nation.French.saluer(..)
```

Sans paramètres

```
public * org.eclipse.nation.French.saluer()
```

Selon le type des paramètres

```
public * org.eclipse.nation.French.saluer(org.eclipse.model.Personne,  
String)
```

Avec les wildcards

```
public * org.eclipse.nation.French.saluer(String, *)
```

Spring

On peut aussi filtrer selon le nombre et/ou le type des paramètres : 0 ou plusieurs paramètres

```
public * org.eclipse.nation.French.saluer(..)
```

Sans paramètres

```
public * org.eclipse.nation.French.saluer()
```

Selon le type des paramètres

```
public * org.eclipse.nation.French.saluer(org.eclipse.model.Personne,  
    String)
```

Avec les wildcards

```
public * org.eclipse.nation.French.saluer(String, *)
```

Ou aussi selon les exceptions

```
public * org.eclipse.nation.French.saluer(..) throws  
    ArithmeticException
```

On peut aussi filtrer selon les annotations

```
@annotation(Override)
```

© Achref EL MOUELHI ©

On peut aussi filtrer selon les annotations

```
@annotation(Override)
```

On peut aussi combiner en utilisant **le** `and` **ou** `or` **ou** `not`

```
bean(*Dao) or @annotation(Repository)
```

Spring

On peut aussi filtrer selon les annotations

```
@annotation(Override)
```

On peut aussi combiner en utilisant **le** `and` **ou** `or` **ou** `not`

```
bean(*Dao) or @annotation(Repository)
```

Ou aussi

```
bean(*Dao) || @annotation(Repository)
```

Spring

Pour factoriser le code précédent, on peut définir un point de coupure

```
public class TestAspect {
    @Pointcut("execution(* org.eclipse.nation.French.saluer(..)")
    public void log() {}

    @Before("log()")
    public void avant() {
        System.out.println("Avant saluer");
    }

    @After("log()")
    public void apres() {
        System.out.println("Après saluer");
    }
}
```

Spring

Pour factoriser le code précédent, on peut définir un point de coupure

```
public class TestAspect {
    @Pointcut("execution(* org.eclipse.nation.French.saluer(..))")
    public void log() {}

    @Before("log()")
    public void avant() {
        System.out.println("Avant saluer");
    }

    @After("log()")
    public void apres() {
        System.out.println("Après saluer");
    }
}
```

En exécutant, le résultat est le même :

```
Avant saluer
Bonjour
Après saluer
```

Exemple avec @Around (le premier paramètre d'une méthode annotée par @Around doit être de type ProceedingJoinPoint)

```
public class TestAspect {
    @Around("execution(* org.eclipse.nation.French.saluer(..)")
    public Object frenchAroundAdvice(ProceedingJoinPoint
        proceedingJoinPoint){
        System.out.println("Avant d'appeler saluer");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("après l'appel de saluer " + value);
        return value;
    }
}
```

Exemple avec @Around (le premier paramètre d'une méthode annotée par @Around doit être de type ProceedingJoinPoint)

```
public class TestAspect {
    @Around("execution(* org.eclipse.nation.French.saluer(..)")
    public Object frenchAroundAdvice(ProceedingJoinPoint
        proceedingJoinPoint){
        System.out.println("Avant d'appeler saluer");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("après l'appel de saluer " + value);
        return value;
    }
}
```

En exécutant, le résultat est :

```
Avant d'appeler saluer
Bonjour
après l'appel de saluer null
```

Spring

Explication

- Tout autre greffon que `@Around` peut déclarer comme premier paramètre `JoinPoint`
- `proceedingJoinPoint.proceed()` : entraine l'exécution de la méthode `saluer()`
- La valeur de retour de `proceedingJoinPoint.proceed()` est la valeur de retour de la méthode `saluer()` (ici, c'est `void`, donc `null`)
- La méthode annotée par `@Around` et `saluer` doivent retourner la même valeur.

Spring

t si la méthode avait des paramètres ?

Il est possible de les récupérer

Modifions l'interface `European` et les deux classes `French` et `English`

```
public interface European {  
    public int saluer(int i);  
}
```

La classe `French`

```
@Component  
public class French implements European{  
    private int rang;  
    public int saluer(int i) {  
        rang = i;  
        System.out.println("Bonjour");  
        return ++i;  
    }  
}
```

Modifions l'interface `European` et les deux classes `French` et `English`

```
public interface European {
    public int saluer(int i);
}
```

La classe `French`

```
@Component
public class French implements European{
    private int rang;
    public int saluer(int i) {
        rang = i;
        System.out.println("Bonjour");
        return ++i;
    }
}
```

La classe `English`

```
@Component
public class English implements European{
    public int saluer(int i) {
        System.out.println("Hello");
        return i;
    }
}
```

Spring

Utiliser `&& args(i)` pour filtrer les méthodes ayant un argument `i`

```
public class TestAspect {
    @Around("execution(* org.eclipse.nation.*nch.saluer(..) && args(i)")
    public Object frenchAroundAdvice(ProceedingJoinPoint proceedingJoinPoint , int i) {
        System.out.println("Valeur du paramètre i dans saluer : " + i);
        System.out.println("Signature : " + proceedingJoinPoint.getSignature());
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("Valeur de retour de saluer : " + value);
        return value;
    }
}
```

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

© AOP

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

Le résultat est :

```
Valeur du paramètre i dans saluer : 4
Signature : int org.eclipse.nation.European.saluer(int)
Bonjour
Valeur de retour de saluer : 5
```

Spring

On peut aussi faire

```
public class TestAspect {
    @Around("execution(* org.eclipse.nation.*nch.saluer(..)")
    public Object frenchAroundAdvice(ProceedingJoinPoint proceedingJoinPoint) {
        System.out.println("Valeur de i dans saluer : " + proceedingJoinPoint.getArgs()[0]);
        System.out.println("Signature : " + proceedingJoinPoint.getSignature());
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("Valeur de retour de saluer : " + value);
        return value;
    }
}
```

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

© AOP

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

Le résultat est :

```
Valeur du paramètre i dans saluer : 4
Signature : int org.eclipse.nation.European.saluer(int)
Bonjour
Valeur de retour de saluer : 5
```

Au lieu d'utiliser `execution` **et préciser des méthodes**

```
execution(* org.eclipse.nation.*nch.saluer(..))
```

© Achref EL MOUËL

Au lieu d'utiliser `execution` et préciser des méthodes

```
execution(* org.eclipse.nation.*nch.saluer(..))
```

On peut utiliser `within` pour dire toutes les méthodes qui sont dans ce package ou un de ses sous-packages

```
within(org.eclipse.nation..*)
```

Spring

Supprimons (ou commentons) les quatre annotations suivantes de la classe `TestAspect`

- `@AspectJ`
- `@EnableAspectJAutoProxy`
- `@Component`
- `@Around(...)`

Nouveau contenu de la classe TestAspect

```
package org.eclipse.aop;

import org.aspectj.lang.ProceedingJoinPoint;

public class TestAspect {

    public Object frenchAroundAdvice(ProceedingJoinPoint
        proceedingJoinPoint) {
        System.out.println("Valeur de parametre i dans saluer : " +
            proceedingJoinPoint.getArgs()[0]);
        System.out.println("Signature : " + proceedingJoinPoint.
            getSignature());
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("Fin de saluer avec une valeur de retour : " +
            value);
        return value;
    }
}
```

Spring

Ajoutons l'espace de nom pour AOP dans `applicationContext.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">
```

Spring

Ajoutons l'espace de nom pour AOP dans `applicationContext.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">
```

Ensuite définissons le bean correspondant à l'aspect `TestAspect`

```
<bean id="testAspect" class="org.eclipse.aop.TestAspect">
</bean>
```

Spring

Définissons le point de coupe et le greffon

```
<aop:config>
  <aop:aspect ref="testAspect">
    <aop:around method="frenchAroundAdvice" pointcut="execution(* org.
      eclipse.nation.*nch.saluer(..) )"></aop:around>
  </aop:aspect>
</aop:config>
```

© Achref EL

Spring

Définissons le point de coupe et le greffon

```
<aop:config>
  <aop:aspect ref="testAspect">
    <aop:around method="frenchAroundAdvice" pointcut="execution(* org.
      eclipse.nation.*nch.saluer(..) )"></aop:around>
  </aop:aspect>
</aop:config>
```

N'oublions pas de scanner les packages contenant les composants

```
<context:component-scan base-package="org.eclipse.nation">
</context:component-scan>
```

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

© Actif

Spring

Et pour tester

```
package org.eclipse.main;

import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        European e = (European) context.getBean("french");
        e.saluer(4);
    }
}
```

Le résultat est :

```
Valeur du paramètre i dans saluer : 4
Signature : int org.eclipse.nation.European.saluer(int)
Bonjour
Valeur de retour de saluer : 5
```