Formation Spring

AOP

Programmation Orientée Aspect

Programmation orientée aspect (AOP)

- ✓ Définition et usage de l'AOP
- ✓ Aspect, Target, Advice, Proxy
- ✓ Expression du Pointcut
- ✓ @Before, @After, @AfterReturning, @AfterThrowing, @Around

Aspect-Oriented Programming

- AOP : **A**spect **O**riented **P**rogrammation (POA en français)
- L'AOP est un paradigme de programmation séparant les considérations techniques (aspect) des descriptions métier
  - Il opère de manière non intrusive
- Les considérations techniques sont souvent transverses à une application
  - Gestion des logs
  - Vérification des paramètres entrants
  - Appel à la sécurité applicative
  - Traitement des exceptions
  - Gestion des transactions

Quels problèmes résout l'AOP ?

- L'AOP n'est pas une technique autonome de conception ou de développement

- L'AOP est complémentaire à l'approche Objet
  - Niveau de réutilisation supérieur à une approche "purement objet"

Quels problèmes résout l'AOP ?

- Un enchevêtrement de code (« code tangling »).

```
public class TransfertServiceImpl implements ITransfertService {
 @Override
 public boolean transfert(double montant) {
  if (!hasPermission(SecurityContext.getPrincipal())) {
   throw new AccessNotGrantedException();
  }
  this.referentiel.stocke(montant);
  return true;
 }
}
```

Quels problèmes résout l'AOP ?

- Un même aspect se trouve éparpillé dans toute l'application (« code scattering »).

```
public class GestionnaireProfilImpl implements IGestionnaireProfil {
 public boolean delete(User utilisateur) {
  if (!hasPermission(SecurityContext.getPrincipal())) {
   throw new AccessNotGrantedException();
  }
  …
 }
}
```

```
public class CompteServiceImpl implements ICompteService {
 public List getComptes() {
  if (!hasPermission(SecurityContext.getPrincipal())) {
   throw new AccessNotGrantedException();
  }
  …
 }
}
```

Premier exemple avec Spring AOP
Configuration XML

- On indique le fait que l'on va faire usage des aspects
- On déclare notre aspect comme bean géré par le Spring

```
<beans …>
    <aop:aspectj-autoproxy/>
    <bean id="profiler" class="com.france.aspect.MonAspect"/>
</beans>
```

Premier exemple avec Spring AOP @Configuration

- On indique le fait que l'on va faire usage des aspects
- On déclare notre aspect comme bean géré par le Spring

```
@Configuration
@EnableAspectJAutoProxy
public class SpringConfigurationAOP {


    @Bean
    public MonAspect monAspect() {
        return new MonAspect();
    }
}
```

Premier exemple avec Spring AOP
Ma classe d'aspect

- L'aspect MonAspect est exécuté avant chaque passage dans une méthode find* de la classe com.MaClasse

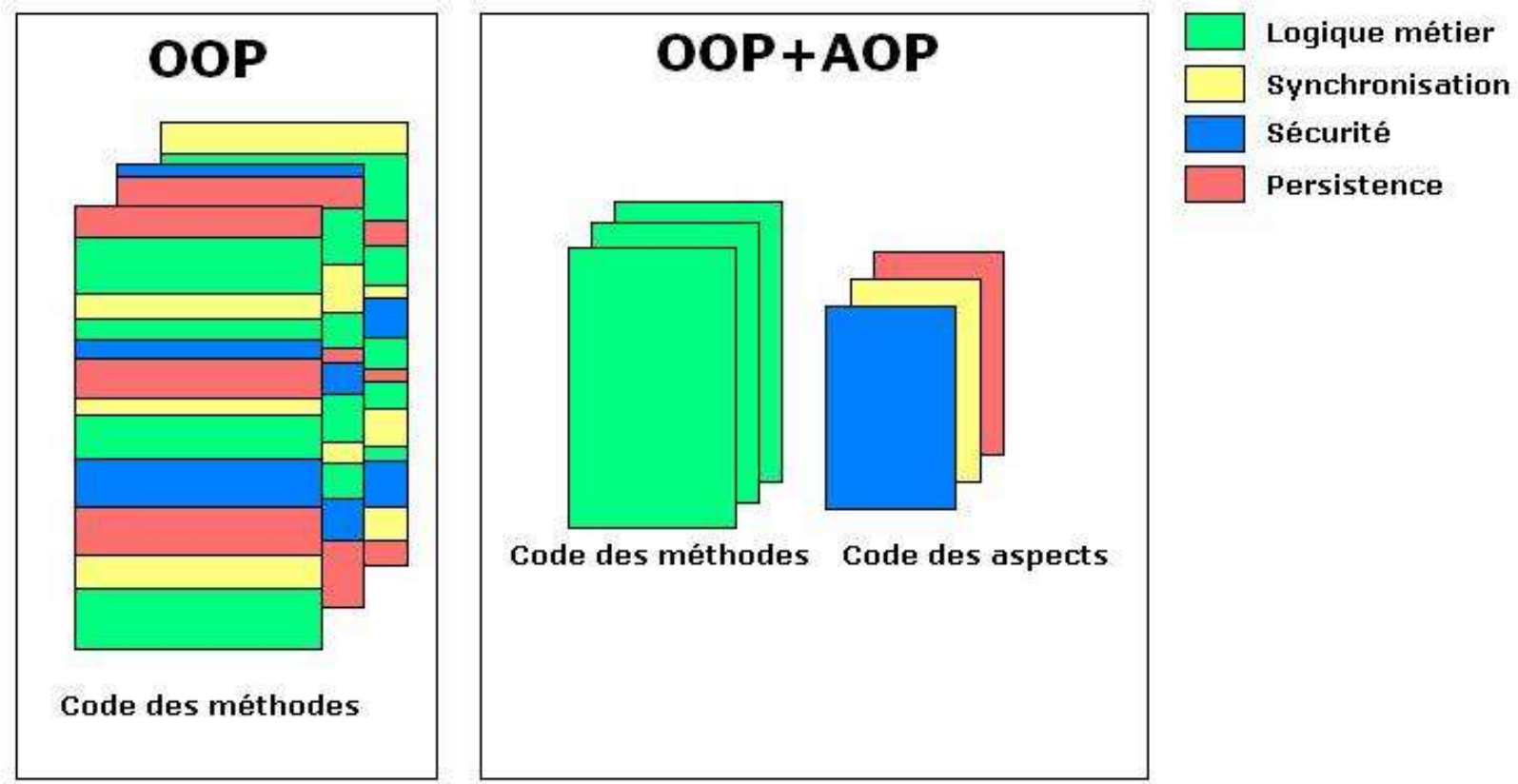
```
@Aspect
public class MonAspect {

    @Before("execution(* com.MaClasse.find*(..))")
    public void processLog(JoinPoint jp)  {
      System.out.println("Avant find ="+jp.getSignature().getName());

    }
}
```

- Remarque : il est aussi possible d'annoter l'aspect avec un stéréotype @Component afin de ne plus avoir à le déclarer comme bean

Quels problèmes résout l'AOP ?



Quels problèmes résout l'AOP ?

- AspectJ
  - Fondation Eclipse - http://www.eclipse.org/aspectj/
  - Technologie initiale (1ère version date de 1995)
  - Langage AOP complet
    - Modifie le byte code pour le tissage.
- SpringAOP
  - Implémentation Java de l'AOP intégrant AspectJ
    - Utilise le mécanisme de **proxy Java** pour le tissage.

Cas d'étude

- Une application contient un ensemble de classes DAO
  - Data Access Objects

- Il s'agit de mesurer l'accès à toutes les méthodes "find" de ces classes
  - Utilisation d'une classe DAOProfiler

Aspect et Target

- **Aspect**
  - L'Aspect est la classe qui contient le code Java à insérer dans la logique de l'application
  - Dans l'exemple précédent, il s'agit de DaoProfiler.

```
@Aspect
public class DAOProfiler {

    …

}
```

Exemple avec Spring (syntaxe @AspectJ)

- **Target**
  - Il s'agit de l'objet visé
  - Un objet Target peut-être visé par plusieurs aspects
  - Dans l'exemple précédent, il s'agit des DAOs
  - L'AOP est non-intrusive : les Targets ne sont pas impactés par la mise en place d'Aspects.

Advice

- **Advice**
  - C'est l'un des traitements à réaliser par l'aspect
  - L'Advice est déclaré dans la classe Aspect
    - Il peut y avoir plusieurs Advices par Aspect
  - Plusieurs catégories :
    - "around", "before", "after returning", "after throwing", "after finally"
    - Exemple de déclaration d'un Advice Before :

```
@Aspect
public class DAOProfiler {
   @Before("execution(* com.MaClasse.find*(..))")
   public void profileFindMethod(JoinPoint jp)  {
   …
   }
}
```

Quelques définitions

- Les concepts de base AOP
  - Aspect
  - Joinpoint (Point de jonction ou d'exécution)
  - Advice (greffon)
  - Pointcut (Point de coupe, d'action, de coupure ou de greffe)
  - Weaving (tissage)

Quelques définitions

- **Joinpoint** (Point d'exécution)
  - Point d'exécution d'un programme tel que l'appel d'une méthode ou l'accès à un membre d'une instance d'objet.
    - Exemple : ClientDAO.findClient, CompteDAO.findCompte

- **Pointcut** (Point d'action)
  - Un pointcut représente un ensemble de joinpoints.
  - La sélection d'un joinpoint s'effectue par filtrage
    - Exemple : execution(* com.MaClasse.find*(..))
      - Dans ce cas, le pointcut représente l'ensemble des méthodes commençant par "find" dans la classe com.MaClasse.

Quelques définitions

- **Advice** (Greffon)
  - Code à exécuter en fonction du joinpoint détecté par le pointcut.

```
@Before ("execution(* find*(..))")
public void profileFindMethod() {…}
```

  - Il existe plusieurs types d'advices
    - « around », « before », « after returning», « after throwing », « after finally ».
  - La précédence est définie par la déclaration des advices dans l'aspect
  - **L'ordre** d'exécution peut être déterminé en implémentant org.springframework.core.**Ordered**.
    - Plus la valeur retournée par Ordered.getOrder() est grande plus la priorité est faible (la priorité est >= 0).

Quelques définitions

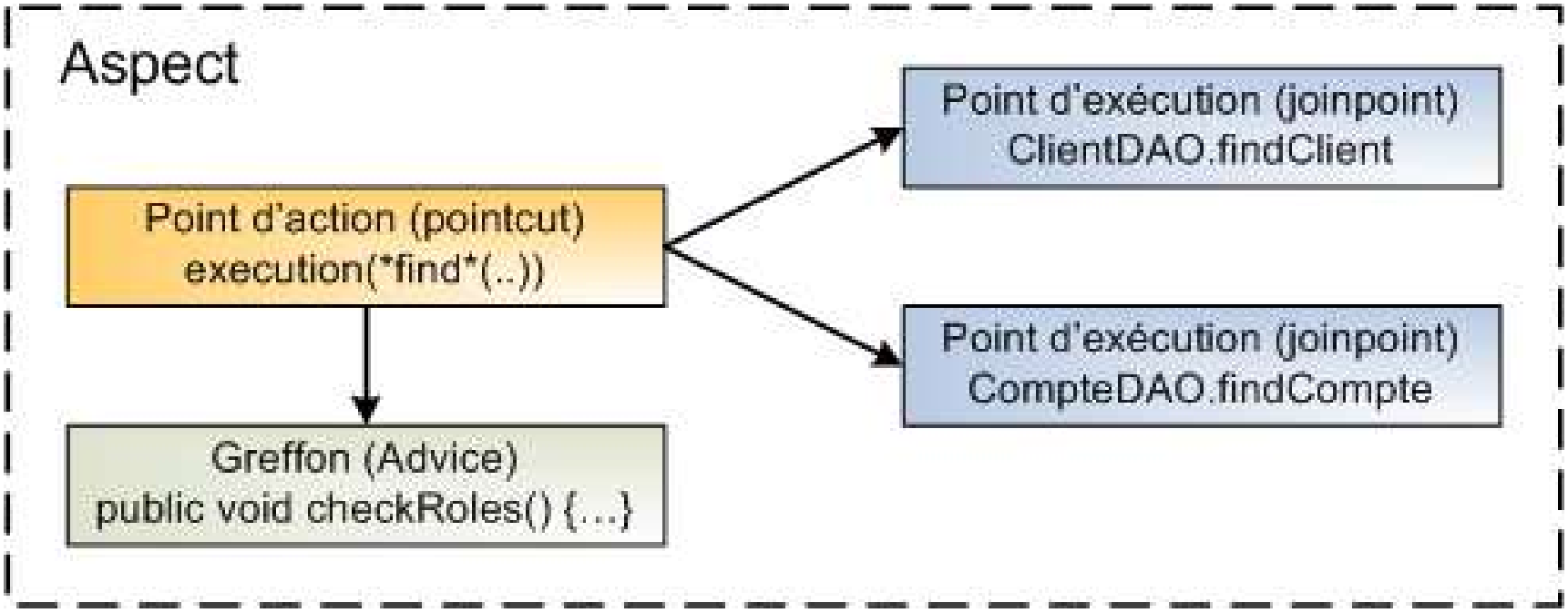
- **Aspect**
  - Classe qui encapsule un ensemble de pointcuts et de joinpoints.

```
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class DaoProfiler {…}
```

- **Weaving** (Tissage)
  - Application d'un ensembles d'aspects au sein d'une application.
  - Le code binaire de l'application contient les instructions implémentées au sein des greffons.

Architecture AOP

- Aspect



Proxy et Weaving

- **Proxy**
  - Classe se substituant à la classe Target
  - Le proxy gère l'appel aux Aspects concernés
  - En Spring AOP, les Proxies sont créés au démarrage
  - 2 approches possibles :
    - Le proxy implémente les mêmes interfaces que la classe
    - Le proxy hérite de la classe Target
- **Weaving**
  - C'est l'action de "tisser" les proxies au démarrage du contexte Spring

Spring AOP peut prendre en compte plusieurs aspects sur une même classe Target

Attention aux erreurs de Cast !

Différences entre Spring AOP et AspectJ

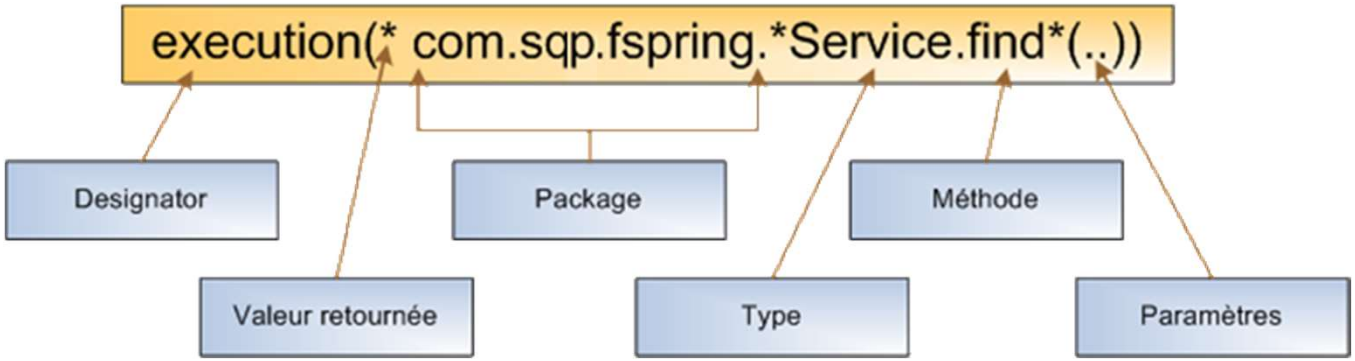
- AspectJ
  - Language de programmation à part entière.
  - Utilise uniquement CGLIB pour le tissage (modification du byte code).
  - Permet le maillage sur les membres d'une classe, publiques ou privés.
  - Travaille sur les points d'exécution privés.

Différences entre Spring AOP et AspectJ

- Spring AOP - Limitations
  - Ne travaille que sur les points d'exécutions publics.
  - Les aspects ne sont applicables **que sur des beans Spring**.
  - Utilise massivement les proxies Java pour le maillage.
    - Applique un proxy dynamique si un point d'exécution est déclaré sur une interface.
    - Si le point d'exécution s'applique sur une classe concrète alors Spring utilise CGLIB pour le tissage.
    - Pour le cas du proxy si une méthode « a » appel une méthode « b » sur la même classe/interface **alors le greffon ne pourra pas s'appliquer sur la méthode « b ».**
  - N'implémente qu'un sous ensemble de l'AOP.

Définir des expressions de pointcut Présentation

- Avec Spring les points d'action sont définis en utilisation les expressions aspectJ
- Définition d'une méthode Java
  - [Modifiers] ReturnType [ClassType] MethodName ([Arguments]) [Throws ExceptionType]
- Exemple d'expression



  - Les caractères « && » (et logique), « || » (ou logique), « ! » (négation) permettent de concaténer des points d'actions.
  - Ce n'est pas une expression régulière !!!

Définir des expressions de pointcut Exemples sur les méthodes (1/2)

- execution(void com.MaClasse.send*(String))
  - Dans la classe MaClasse du package com, toute méthode commençant par « send », prenant en paramètre de type String et ne retournant aucune valeur.

- execution(* com.MaClasse.send(*))
  - Dans la classe MaClasse du package com, toute méthode nommée « send » n'acceptant **qu'un seul** paramètre (peut importe son type).

- execution(* com.MaClasse.send(int, ..))
  - Dans la classe MaClasse du package com, toute méthode nommée « send », prenant comme premier paramètre un entier primitif. Les caractères « .. » signifient **0 ou plusieurs** paramètres.

- execution(void exemple.MessageServiceImpl.*(..))
  - Toute méthode de la classe « MessageServiceImpl » du package exemple.

Définir des expressions de pointcut Exemples sur les méthodes (2/2)

- execution(void exemple.MessageService+.send(*))
  - Toute méthode nommée « send » de n'importe quel objet de type MessageService incluant les sous classes ou implémentations de « MessageService ».

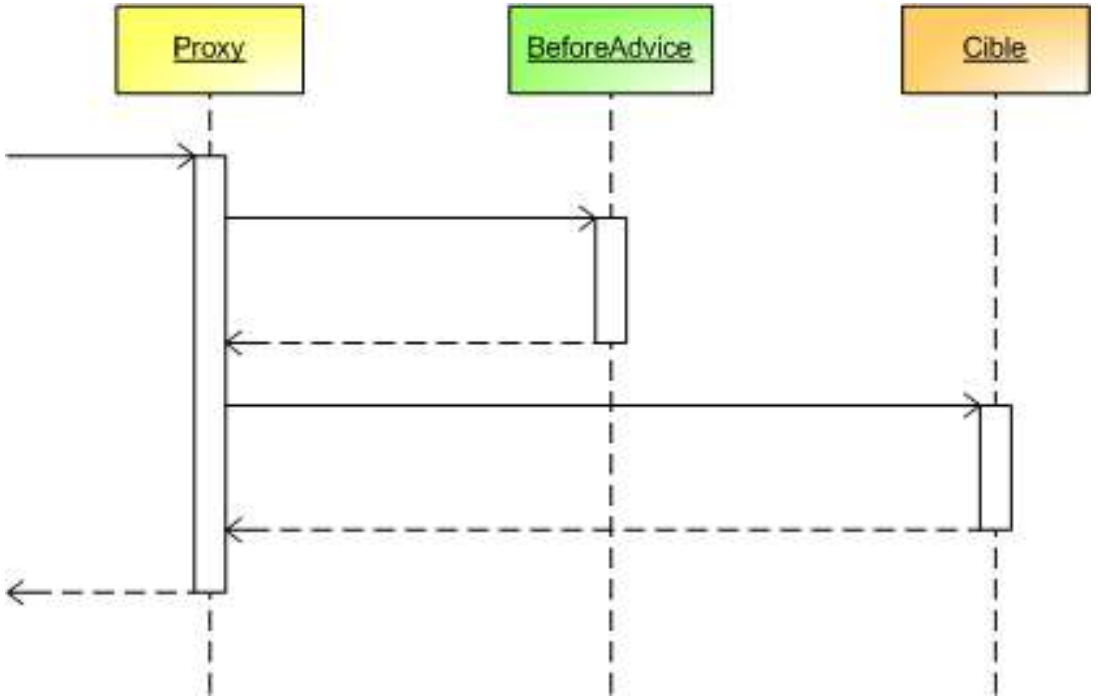
- execution(@javax.annotation.security.RolesAllowed void send*(*))
  - Toute méthode commençant par « send », annotée par « @RolesAllowed ».

Définir des expressions de pointcut Exemples sur les packages

- execution(* france.*.exemple.*.*(..))
  - Un seul répertoire entre « france » et « exemple ».
- execution(* france..exemple.*.*(..))
  - Un ou plusieurs répertoires entre « france » et « exemple ».
- execution(* *..exemple.*.*(..))
  - N'importe quel sous package « exemple ».

Les types de d'advices
Before (1/2)

- L'advice est exécuté avant l'appel de la méthode cible



Les types de d'advices
Before (2/2)

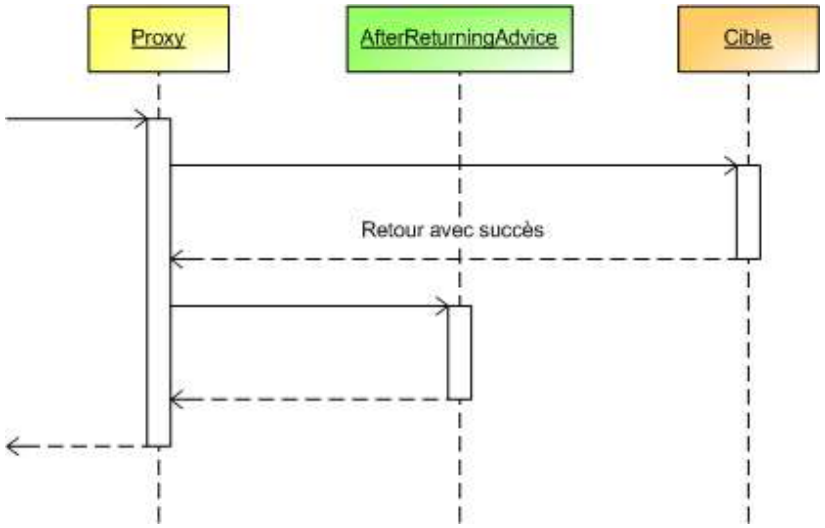
- Si l'advice lève une exception, la classe cible ne sera pas appelée
- Exemple : tracer les appels aux méthodes « set » dans la classe MaClasse du package *com*

```
@Aspect
public class PropertyTracker {
 private static final Logger logger =
Logger.getLogger(PropertyTracker.class);

 @Before("execution(void com.MaClasse.set*(..))")
 public void trackChange() {
  logger.info("Property will change...");
 }
}
```

Les types de d'advices
After Returning

- Exécute l'advice après que la méthode cible ait terminé son traitement (sans lever d'exception)

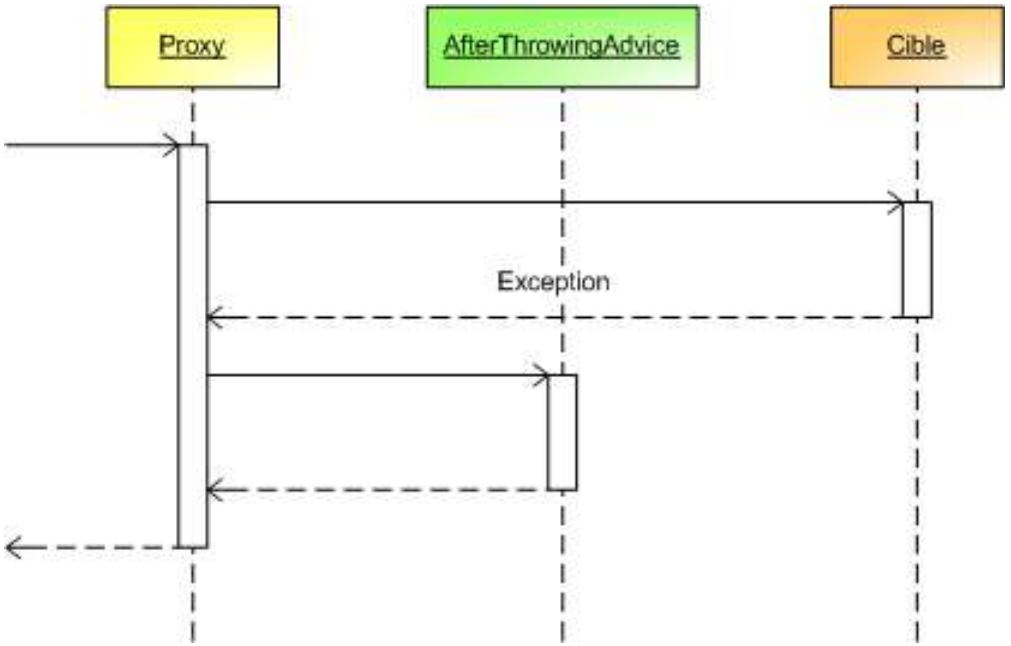


- Exemple : ici détecte toutes les méthodes renvoyant un objet de type « Event »

```
@AfterReturning(value="execution(* service..*.*(..))", returning="unEvent")
public void trackEvents(JoinPoint jp, Object unEvent) {
  logger.info(jp.getSignature() + " retourne l'instance " + unEvent.toString());
}
```

Les types de d'advices
After Throwing (1/2)

- Exécute l'advice si la méthode cible lève une exception



Les types de d'advices
After Throwing (2/2)

- Exemple : envoi d'un email lors de la levée d'une exception de type « DataAccessException »

```
@AfterThrowing(value="execution(* *..Repository+.*(..))", throwing="e")
public void report(DataAccessException e) {
      mailService.sendFailureEmail("Exception in database " + e);
}
```

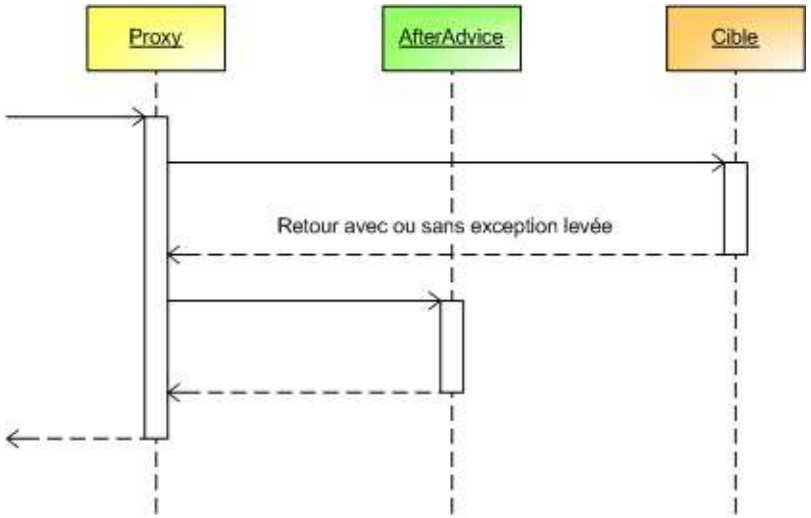
  - Il est possible de définir une même méthode en indiquant un autre type d'exception.

- L'advice ne peut arrêter la propagation de l'exception
  - En revanche, il peut substituer le type d'exception.

```
@AfterThrowing(value="execution(* *..Repository+.*(..))", throwing="e")
public void report(DataAccessException e) {
      mailService.sendFailureEmail("Exception in database " + e);
      throw new GenericDatabaseException(e);
}
```

Les types de d'advices
After

- L'advice est appelé même si la méthode cible a levée une exception

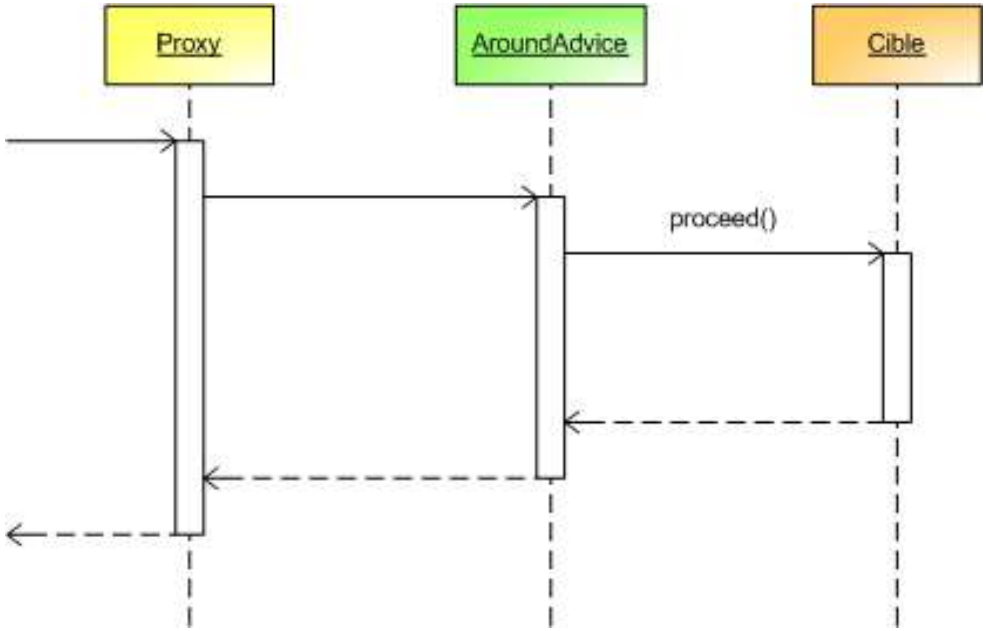


- Exemple : trace toutes les mises à jours

```
@After("execution(void com.MaClasse.update*(*))")
public void trackUpdates() {
      logger.info("Database info has changed...");
}
```

Les types de d'advices
Around (1/3)

- « Around » s'insère dans le flux d'exécution
  - Permet de lancer des tâches avant et après l'exécution de la méthode cible.



Les types de d'advices
Around (2/3)

- Exemple :

```
@Around("execution(* exemple.service.*.*(..))")
public Object cache(ProceedingJoinPoint pj) throws Throwable {
  // Code avant votre méthode
  // ...
  // Exécution de la vraie méthode et récupération du résultat
  Object value = pj.proceed();
  // Code après votre méthode
  // ...
  // Il faut retourner le résultat
  return value;
}
```

Les types de d'advices Around (3/3)

- Passage de paramètres à l'advice
  - Préciser dans la déclaration de l'advice : args() et dans la signature du pointcut le type de l'argument

```
@Around("execution(int exemple.service.MonService.maMethode(String, String)) && args(param1, param2)")
public int cache(ProceedingJoinPoint pj, String param1, String param2) throws Throwable {
  int value = -1;
  // ...
  return value;
}
```

  - Vous pouvez ainsi modifier les paramètres et appeler la 'vraie' méthode via la méthode proceed(Object[]) de ProceedingJoinPoint

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 09 : Un aspect de log