Intercepteurs

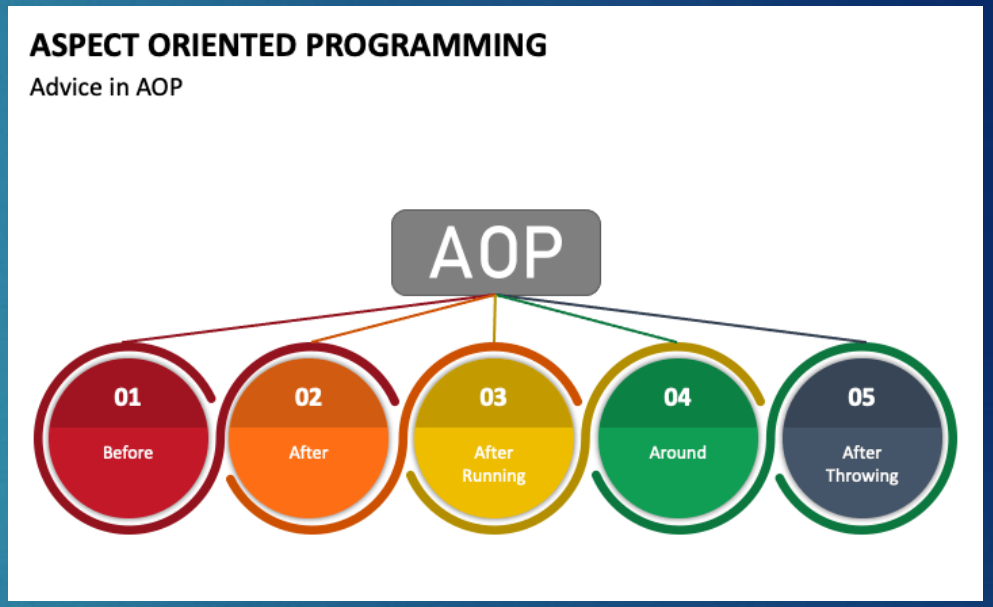
- Interfaces AroundAdvice et Interceptor
- Modifier le comportement des méthodes
- Traitement avant et/ou après l'appel
- Chacun possède son annotation

```
public interface AroundAdvice {
  void before(Object instance, Method method, Object[] args) throws Throwable;
  void after(Object instance, Method method, Object[] args, Object result) throws Throwable;
}

@FunctionalInterface
public interface Interceptor {
  Object intercept(Object instance, Method method, Object[] args, Invocation invocation) throws Throwable;
}
```

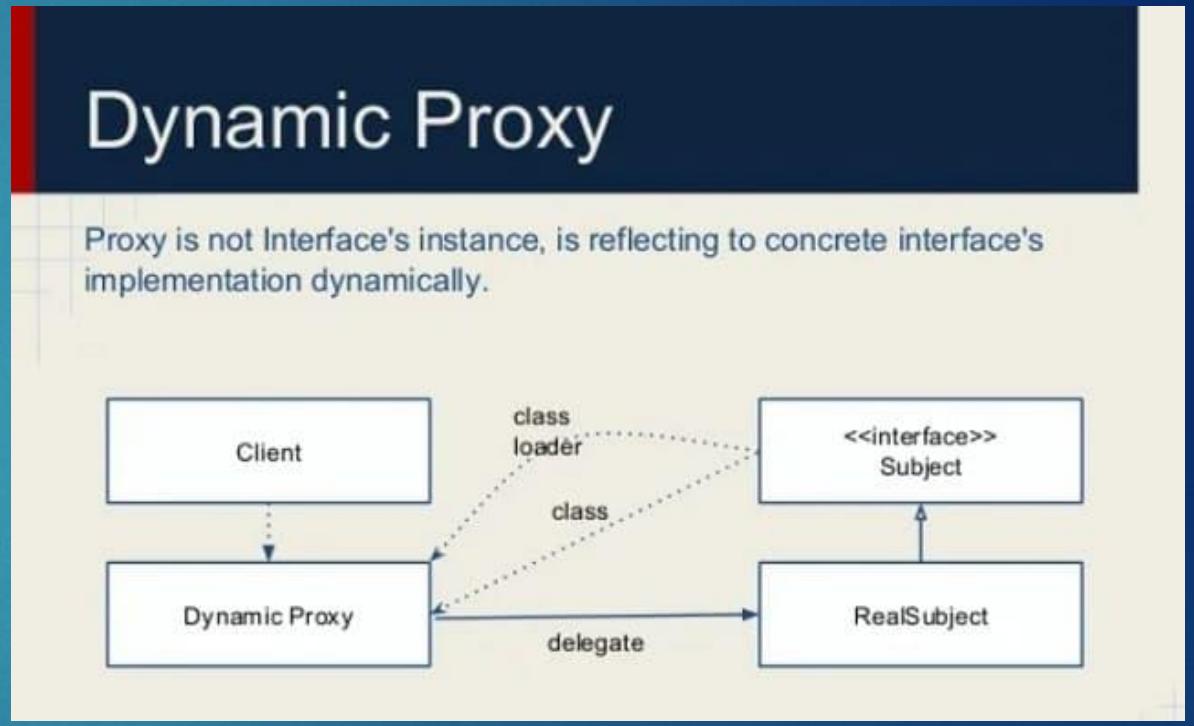
Aspect Oriented Programming (AOP)

- Séparation du code métier et du code transversal à plusieurs parties de l'appli
- Exemples de code transversal : logging, monitoring, sécurité
- Aspect = classe qui encapsule le comportement transversal
- Points de jonction en Java : appel de méthode, constructeur, propriétés (getters et setters)



Proxy dynamique

- Implémenter des interfaces à l'exécution
- Création dynamique d'objet
- Objectif : Manipuler les méthodes sans modifier le code source
- Cas d'utilisation des intercepteurs
- APIs du JDK : java.lang.reflect.Proxy



Exemple de proxy simple


```
interface Service {void performAction();}

class RealService implements Service {
  @Override
  public void performAction() {
    System.out.println("Action effectuée par le service réel");
}}

class ProxyHandler implements InvocationHandler {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
    System.out.println("Avant l'appel de la méthode...");
    var result = method.invoke(target, args);  // Appel de la méthode
    System.out.println("Après l'appel de la méthode...");
    return result;
 }}
```

```
// Main :
public static void main(String[] args) {
  var  service = new RealService();
   var proxyInstance =
   (Service)   Proxy.newProxyInstance(

realService.getClass().getClassLoader(),
           new Class[]{Service.class},
           new ProxyHandler(realService)
    );
    proxyInstance.performAction();
  }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-23.
Avant l'appel de la méthode...
Action effectuée par le service réel
Après l'appel de la méthode...

Process finished with exit code 0
```

Invocation


```
package java.lang.reflect;

@FunctionalInterface
public interface InvocationHandler {
  Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
}
```

```
@FunctionalInterface
public interface Invocation {
  Object proceed(Object instance, Method method, Object[] args) throws Throwable;
}
```

- Utilisation d'une interface similaire à InvocationHandler du JDK
- Intercepteurs : avant ET/OU après méthode
- Invocateurs : appel réel de la méthode
- Instance = this de l'objet, method = méthode appelée, args = paramètres de la méthode
- API de réflexion dans le package java.lang.reflect (cf. TP2)

Bidouilleries en Java


```
advice.before(instance, method, args);
Object result = null;
try {
  result = invocation.proceed(instance, method, args);
} finally {
  advice.after(instance, method, args, result);
}
```

- Cas rare d'initialisation à null
- Bloc try/catch/finally : finally toujours exécuté peu importe le résultat
- Création d'un cache plus simple avec une table de hachage
- Generics imbriqués

```
private final HashMap<Class<?>, List<Interceptor>> interceptorMap;
private final HashMap<Method, Invocation> cache;

. . .

public void addInterceptor(annotationClass, interceptor) {
  interceptorMap.computeIfAbsent(annotationClass, _ -> new ArrayList<>()).add(interceptor);
  cache.clear();
}
```

Supplément streams

```
List<Interceptor> findInterceptors(Method method) {
  return Stream.of(
      Arrays.stream(method.getDeclaringClass().getAnnotations()),
      Arrays.stream(method.getAnnotations()),
      Arrays.stream(method.getParameterAnnotations()).flatMap(Arrays::stream))
    .flatMap(s -> s)
    .map(Annotation::annotationType)
    .distinct()
    .flatMap(annotationType -> interceptorMap.getOrDefault(annotationType, List.of()).stream())
    .toList();
}
```

- flatMap : stream vers steam (aplatit en un seul flux)
- Map : transformation sur chaque élément, sans changer la structure initiale

Intérêts de flatMap :

- Manipuler des structures de données de dimension >= 2
- Concaténer plusieurs streams en un seul (cf. ci-dessus)