Spring AOP and Method Proxies

jim stafford

Fall 2022 v2020-07-18: Built: 2022-12-07 06:17 EST

Table of Contents

Chapter 1. Introduction

Many times, business logic must execute additional behavior that is outside of its core focus. For example, auditing, performance metrics, transaction control, retry logic, etc. We need a way to bolt on additional functionality ("advice") without knowing what the implementation code ("target") will be, what interfaces it will implement, or even if it will implement an interface.

Frameworks must solve this problem every day. To fully make use of advanced frameworks like Spring and Spring Boot, it is good to understand and be able to implement solutions using some of the dynamic behavior available like:

- Java Reflection
- Dynamic (Interface) Proxies
- CGLIB (Class) Proxies
- Aspect Oriented Programming (AOP)

1.1. Goals

You will learn:

- to decouple potentially cross-cutting logic away from core business code
- to obtain and invoke a method reference
- to wrap add-on behavior to targets in advice
- to construct and invoke a proxy object containing a target reference and decoupled advice
- to locate callable join point methods in a target object and apply advice at those locations

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. obtain a method reference and invoke it using Java Reflection
2. create a JDK Dynamic Proxy to implement adhoc interface(s) to form a proxy at runtime for implementing advice
3. create a CGLIB Proxy to dynamically create a subclass to form a proxy at runtime for implementing advice
4. implement dynamically assigned behavior to methods using Spring Aspect-Oriented Programming (AOP) and AspectJ
5. identify method join points to inject using pointcut expressions
6. implement advice that executes before, after, and around join points
7. implement parameter injection into advice

Chapter 2. Rationale

Our problem starts off with two independent classes depicted as `ClassA` and `ClassB` and a caller labelled as `Client`. `doSomethingA()` is unrelated to `doSomethingB()` but may share some current or future things in common — like transactions, database connection, or auditing requirements.

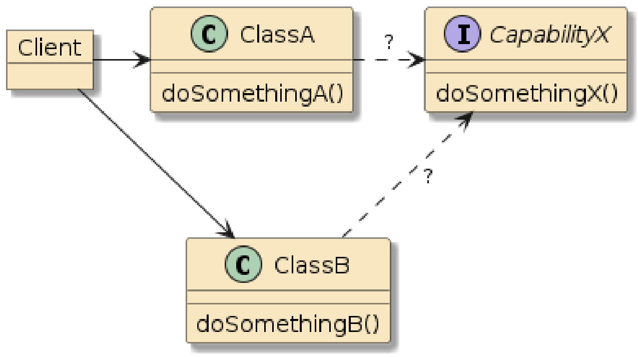


Figure 1. New Cross-Cutting Design Decision

We come to a point where `CapabilityX` is needed in both `doSomethingA()` and `doSomethingB()`. An example of this could be normalization or input validation. We could implement the capability within both operations or in near-best situations implement a common solution and have both operations call that common code.

Reuse is good, but depending on how you reuse may get you in trouble.

2.1. Adding More Cross-Cutting Capabilities

Of course, it does not end there and we have established what could be a bad pattern of coupling the core business code of `doSomethingA()` and `doSomethingB()` with tangential features of the additional capabilities (e.g., auditing, timing, retry logic, etc.).

What other choice do we have?

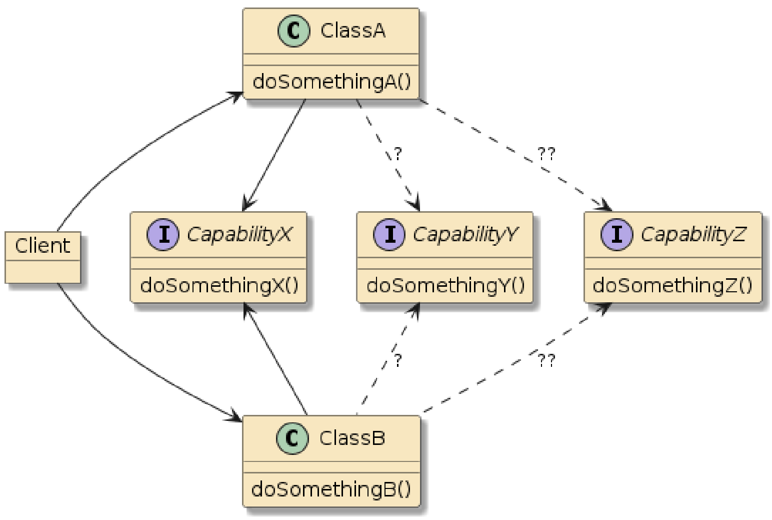
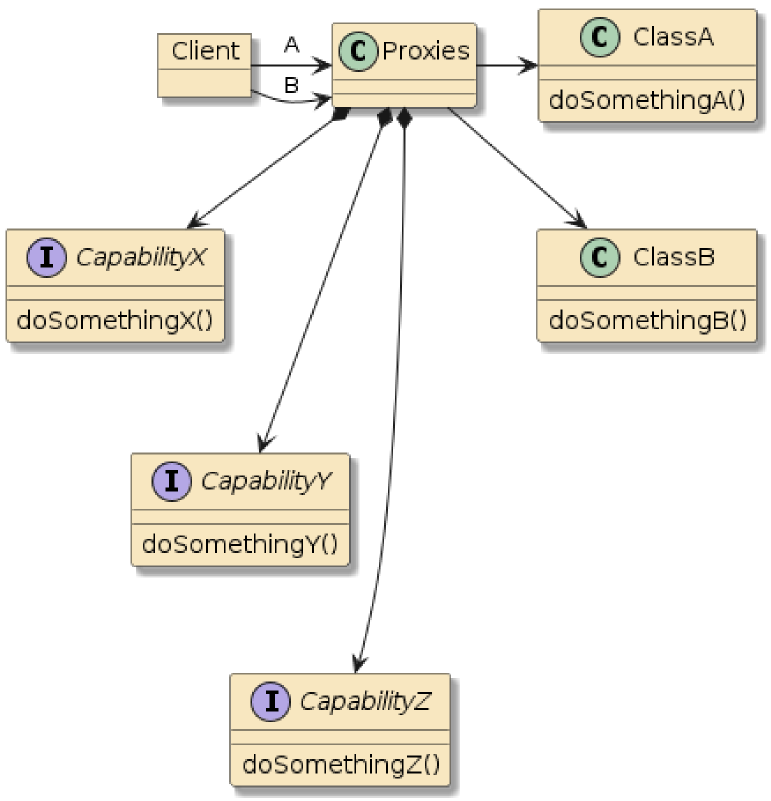


Figure 2. More Cross-Cutting Capabilities

2.2. Using Proxies



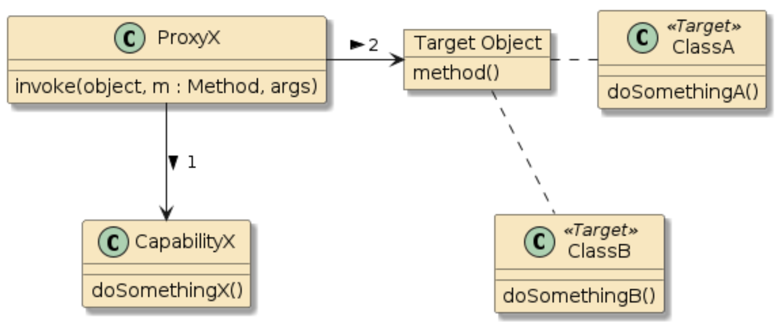
What we can do instead is leave `ClassA` and `ClassB` alone and wrap calls to them in a series of one or more proxied calls. `X` might perform auditing, `Y` might perform normalization of inputs, and `Z` might make sure the connection to the database is available and a transaction active. All we need `ClassA` and `ClassB` to do is their designated "something".

However, there is a slight flaw to overcome. `ClassA` and `ClassB` do not implement a common interface; `doSomethingA()` and `doSomethingB()` look very different in signature, and capabilities; neither `X`, `Y`, `Z` or any of the proxy layers know a thing about `ClassA` or `ClassB`.

We need to tie these unrelated parts together. Lets begin to solve this with Java Reflection.

Chapter 3. Reflection

Java Reflection provides a means to examine a Java class and determine facts about it that can be useful in describing it and invoking it.



Lets say I am in ProxyX applying doSomethingX() to the call and I want to invoke some to-be-determined (TBD) method in the target object. ProxyX does not need to know anything except what to call to have the target invoked. This call will eventually point to doSomethingA() or doSomethingB() at some point.

We can use Java Reflection to solve this problem by

- inspecting the target object's class (ClassA or ClassB) to obtain a reference to the method (doSomethingA() or doSomethingB()) we wish to call
- identify the arguments to be passed to the call
- identify the target object to call

Let's take a look at this in action.

3.1. Reflection Method

Java Reflection provides the means to obtain a handle to Fields and Methods of a class. In the example below, I show code that obtains a reference to the createItem method, in the ItemsService interface, and accepting objects of type ItemDTO.

Obtaining a Java Reflection Method Reference

```
import info.ejava.examples.svc.aop.items.services.ItemsService;
import java.lang.reflect.Method;
...
Method method = ItemsService.class.getMethod("createItem", ItemDTO.class); ①
log.info("method: {}", method);
...
```

① getting reference to method within ItemsService interface

Java Class has numerous methods that allow us to inspect interfaces and classes for fields, methods, annotations, and related types (e.g., inheritance). getMethod() looks for a method with the String name ("createItem") provided that accepts the supplied type(s) (ItemDTO). Arguments is a vararg array, so we can pass in as many types as necessary to match the intended call.

The result is a Method instance that we can use to refer to the specific method to be called — but not the target object or specific argument values.

Example Reflection Method Output

```
method: public abstract info.ejava.examples.svc.aop.items.dto.ItemDTO
    info.ejava.examples.svc.aop.items.services.ItemsService.createItem(
        info.ejava.examples.svc.aop.items.dto.ItemDTO)
```

3.2. Calling Reflection Method

We can invoke the Method reference with a target object and arguments and receive the response as a `java.lang.Object`.

Example Reflection Method Call

```
import info.ejava.examples.svc.aop.items.dto.BedDTO;
import info.ejava.examples.svc.aop.items.services.ItemsService;
import java.lang.reflect.Method;
...

ItemsService<BedDTO> bedsService = ... ①
Method method = ...

//invoke method using target object and args
Object[] args = new Object[] { BedDTO.bedBuilder().name("Bunk Bed").build() }; ②
log.info("invoke calling: {}({})", method.getName(), args);

Object result = method.invoke(bedsService, args); ③

log.info("invoke {} returned: {}", method.getName(), result);
```

① we must obtain a target object to invoke

② arguments are passed into `invoke()` using a varargs array

③ invoke the method on the object and obtain the result

Example Method Reflection Call Output

```
invoke calling: createItem([BedDTO(super=ItemDTO(id=0, name=Bunk Bed))])
invoke createItem returned: BedDTO(super=ItemDTO(id=1, name=Bunk Bed))
```

3.3. Reflection Method Result

The end result is the same as if we called the `BedsServiceImpl` directly.

Example Method Reflection Result

```
    //obtain result from invoke() return
    BedDTO createdBed = (BedDTO) result;
    log.info("created bed: {}", createdBed);----
```

Example Method Reflection Result Output

```
created bed: BedDTO(super=ItemDTO(id=1, name=Bunk Bed))
```

There, of course, is more to Java Reflection that can fit into a single example — but lets now take that fundamental knowledge of a `Method` reference and use that to form some more encapsulated proxies using JDK Dynamic (Interface) Proxies and CGLIG (Class) Proxies.

Chapter 4. JDK Dynamic Proxies

The JDK offers a built-in mechanism for creating dynamic proxies for interfaces. These are dynamically generated classes — when instantiated at runtime — will be assigned an arbitrary set of interfaces to implement. This allows the generated proxy class instances to be passed around in the application, masquerading as the type(s) they are a proxy for. This is useful in frameworks to implement features for implementation types they will have no knowledge of until runtime. This eliminates the need for compile-time generated proxies. [1]

4.1. Creating Dynamic Proxy

We create a JDK Dynamic Proxy using the static `newProxyInstance()` method of the `java.lang.reflect.Proxy` class. It takes three arguments: the classloader for the supplied interfaces, the interfaces to implement, and handler to implement the custom advice details of the proxy code and optionally complete the intended call (e.g., security policy check handler).

In the example below, `GrillServiceImpl` extends `ItemsServiceImpl<T>`, which implements `ItemsService<T>`. We are creating a dynamic proxy that will implement that interface and delegate to an advice instance of `MyInvocationHandler` that we write.

Creating Dynamic Proxy

```
import info.ejava.examples.svc.aop.items.aspects.MyDynamicProxy;
import info.ejava.examples.svc.aop.items.services.GrillsServiceImpl;
import info.ejava.examples.svc.aop.items.services.ItemsService;
import java.lang.reflect.Proxy;
...

ItemsService<GrillDTO> grillService = new GrillsServiceImpl(); ①

ItemsService<GrillDTO> grillServiceProxy = (ItemsService<GrillDTO>)
    Proxy.newProxyInstance( ②
            grillService.getClass().getClassLoader(),
            new Class[]{ItemsService.class}, ③
            new MyInvocationHandler(grillService) ④
        );

log.info("created proxy {}", grillServiceProxy.getClass());
log.info("handler: {}",
        Proxy.getInvocationHandler(grillServiceProxy).getClass());
log.info("proxy implements interfaces: {}",
        ClassUtils.getAllInterfaces(grillsServiceProxy.getClass()));
```

① create target implementation object unknown to dynamic proxy

② instantiate dynamic proxy instance and underlying dynamic proxy class

③ identify the interfaces implemented by the dynamic proxy class

④ provide advice instance that will handle adding proxy behavior and invoking target instance

4.2. Generated Dynamic Proxy Class Output

The output below shows the `$Proxy86` class that was dynamically created and that it implements the `ItemsService` interface and will delegate to our custom `MyInvocationHandler` advice.

Example Generated Dynamic Proxy Class Output

```
created proxy: class com.sun.proxy.$Proxy86
handler: class info.ejava.examples.svc.aop.items.aspects.MyInvocationHandler
proxy implements interfaces:
  [interface info.ejava.examples.svc.aop.items.services.ItemsService, ①
   interface java.io.Serializable] ②
```

① `ItemService` interface supplied at runtime

② `Serializable` interface implemented by DynamicProxy implementation class

4.3. Alternative Proxy All Construction

Alternatively, we can write a convenience builder that simply forms a proxy for all implemented interfaces of the target instance. The Apache Commons ClassUtils utility class is used to obtain a list of all interfaces implemented by the target object's class and parent classes.

Alternative Proxy All Construction

```
import org.apache.commons.lang3.ClassUtils;
...
@RequiredArgsConstructor
public class MyInvocationHandler implements InvocationHandler {
    private final Object target;

    public static Object newInstance(Object target) {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
            ClassUtils.getAllInterfaces(target.getClass()).toArray(new Class[0]),①
            new MyInvocationHandler(target));
    }
```

① Apache Commons ClassUtils used to obtain all interfaces for target object

4.4. InvocationHandler Class

JDK Dynamic Proxies require an instance that implements the `InvocationHandler` interface to implement the custom work (aka "advice") and delegate the call to the target instance (aka "around advice"). This is a class that we write. The `InvocationHandler` interface defines a single reflection-oriented `invoke()` method taking the proxy, method, and arguments to the call. Construction of this object is up to us — but the raw target object is likely a minimum requirement — as we will need that to make a clean, delegated call.

Example Dynamic Proxy InvocationHandler

```
...
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
...
@RequiredArgsConstructor
public class MyInvocationHandler implements InvocationHandler { ①
    private final Object target; ②

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable { ③
        //proxy call
    }
}
```

① class must implement `InvocationHandler`

② raw target object to invoke

③ `invoke()` is provided reflection information for call

4.5. InvocationHandler invoke() Method

The `invoke()` method performs any necessary advice before or after the proxied call and uses standard method reflection to invoke the target method. You should recall the `Method` class from the earlier discussion on Java Reflection. The response or thrown exception can be directly returned or thrown from this method.

InvocationHandler Proxy invoke() Method

```
@Override
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
    //do work ...
    log.info("invoke calling: {}({})", method.getName(), args);

    Object result = method.invoke(target, args);

    //do work ...
    log.info("invoke {} returned: {}", method.getName(), result);
    return result;
}
```

Must invoke raw target instance — not the proxy

Calling the supplied proxy instance versus the raw target instance would result in a circular loop. We must somehow have a reference to the raw target to be able to directly invoke that instance.

4.6. Calling Proxied Object

The following is an example of the proxied object being called using its implemented interface.

Example Proxied Object Call

```
GrillDTO createdGrill = grillServiceProxy.createItem(
        GrillDTO.grillBuilder().name("Broil King").build());
log.info("created grill: {}", createdGrill);
```

The following shows that the call was made to the target object, work was able to be performed before and after the call within the `InvocationHandler`, and the result was passed back as the result of the proxy.

Example Proxied Call Output

```
invoke calling: createItem([GrillDTO(super=ItemDTO(id=0, name=Broil King))]) ①
invoke createItem returned: GrillDTO(super=ItemDTO(id=1, name=Broil King)) ②
created grill: GrillDTO(super=ItemDTO(id=1, name=Broil King)) ③
```

① work performed within the `InvocationHandler` advice prior to calling target

② work performed within the `InvocationHandler` advice after calling target

③ target method’s response returned to proxy caller

JDK Dynamic Proxies are definitely a level up from constructing and calling `Method` directly as we did with straight Java Reflection. They are the proxy type of choice within Spring but have the limitation that they can only be used to proxy interface-based objects and not no-interface classes.

If we need to proxy a class that does not implement an interface — CGLIB is an option.

[1] *"Dynamic Proxy Classes"*, Oracle JavaSE 8 Technotes

Chapter 5. CGLIB

Code Generation Library (CGLIB) is a byte instrumentation library that allows the manipulation or creation of classes at runtime. [1]

Where JDK Dynamic Proxies implement a proxy behind an interface, CGLIB dynamically implements a sub-class of the class proxied.

This library has been fully integrated into `spring-core`, so there is nothing additional to add to begin using it directly (and indirectly when we get to Spring AOP).

5.1. Creating CGLIB Proxy

The following code snippet shows a CGLIB proxy being constructed for a `ChairsServiceImpl` class that implements no interfaces. Take note that there is no separate target instance — our generated proxy class will be a subclass of `ChairsServiceImpl` and it will be part of the target instance. The real target will be in the base class of the instance. We register an instance of `MethodInterceptor` to handle the custom advice and optionally complete the call. This is a class that we write when authoring CGLIB proxies.

Creating CGLIB Proxy

```
...
import info.ejava.examples.svc.aop.items.aspects.MyMethodInterceptor;
import info.ejava.examples.svc.aop.items.services.ChairsServiceImpl;
import org.springframework.cglib.proxy.Enhancer;
...
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(ChairsServiceImpl.class); ①
enhancer.setCallback(new MyMethodInterceptor()); ②
ChairsServiceImpl chairsServiceProxy = (ChairsServiceImpl)enhancer.create(); ③

log.info("created proxy: {}", chairsServiceProxy.getClass());
log.info("proxy implements interfaces: {}",
        ClassUtils.getAllInterfaces(chairsServiceProxy.getClass()));
```

① create CGLIB proxy as sub-class of target class

② provide instance that will handing adding proxy advice behavior and invoking base class

③ instantiate CGLIB proxy — this is our target object

The following output shows that the proxy class is of a CGLIB proxy type and implements no known interface other than the CGLIB `Factory` interface. Note that we were able to successfully cast this proxy to the `ChairsServiceImpl` type — the assigned base class of the dynamically built proxy class.

Example Generated CGLIB Proxy Class

```
created proxy: class
info.ejava.examples.svc.aop.items.services.GrillsServiceImpl$$EnhancerByCGLIB$$a4035db
```

```
5
proxy implements interfaces: [interface org.springframework.cglib.proxy.Factory] ①
```

① Factory interface implemented by CGLIB proxy implementation class

5.2. MethodInterceptor Class

To intelligently process CGLIB callbacks, we need to supply an advice class that implements `MethodInterceptor`. This gives us access to the proxy instance being invoked, the reflection `Method` reference, call arguments, and a new type of parameter — `MethodProxy`, which is a reference to the target method implementation in the base class.

Example MethodInterceptor Class

```
...
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class MyMethodInterceptor implements MethodInterceptor {
    @Override
    public Object intercept(Object proxy, Method method, Object[] args,
                                           MethodProxy methodProxy) ①
                                           throws Throwable {
        //proxy call
    }
}
```

① additional method used to invoke target object implementation in base class

5.3. MethodInterceptor intercept() Method

The details of the `intercept()` method are much like the other proxy techniques we have looked at and will look at in the future. The method has a chance to do work before and after calling the target method, optionally calls the target method, and returns the result. The main difference is that this proxy is operating within a subclass of the target object.

Example MethodInterceptor intercept() Method

```
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;
...
@Override
public Object intercept(Object proxy, Method method, Object[] args,
        MethodProxy methodProxy) throws Throwable {
    //do work ...
    log.info("invoke calling: {}({})", method.getName(), args);

    Object result = methodProxy.invokeSuper(proxy, args); ①
    //do work ...
    log.info("invoke {} returned: {}", method.getName(), result);

    //return result
    return result;
}
```

① invoking target object implementation in base class

5.4. Calling CGLIB Proxied Object

The net result is that we are still able to reach the target object's method and also have the additional capability implemented around the call of that method.

Example CGLIB Proxied Object Call

```
ChairDTO createdChair = chairsServiceProxy.createItem(
        ChairDTO.chairBuilder().name("Recliner").build());
log.info("created chair: {}", createdChair);
```

Example CGLIB Proxied Object Call Output

```
invoke calling: createItem([ChairDTO(super=ItemDTO(id=0, name=Recliner))])
invoke createItem returned: ChairDTO(super=ItemDTO(id=1, name=Recliner))
created chair: ChairDTO(super=ItemDTO(id=1, name=Recliner))
```

[1] *"Introduction to cglib"*, Baeldung, Aug 2019

Chapter 6. Interpose

OK — all that dynamic method calling was interesting, but what sets all that up? Why do we see proxies sometimes and not other times in our Spring application? We will get to the setup in a moment, but lets first address when can we expect this type of behavior magically setup for us and not. What occurs automatically is primarily a matter of "interpose". Interpose is a term used when we have a chance to insert a proxy in between the caller and target object. The following diagram depicts three scenarios: buddy methods of same class, calling method of manually instantiated class, and calling method of injected object.

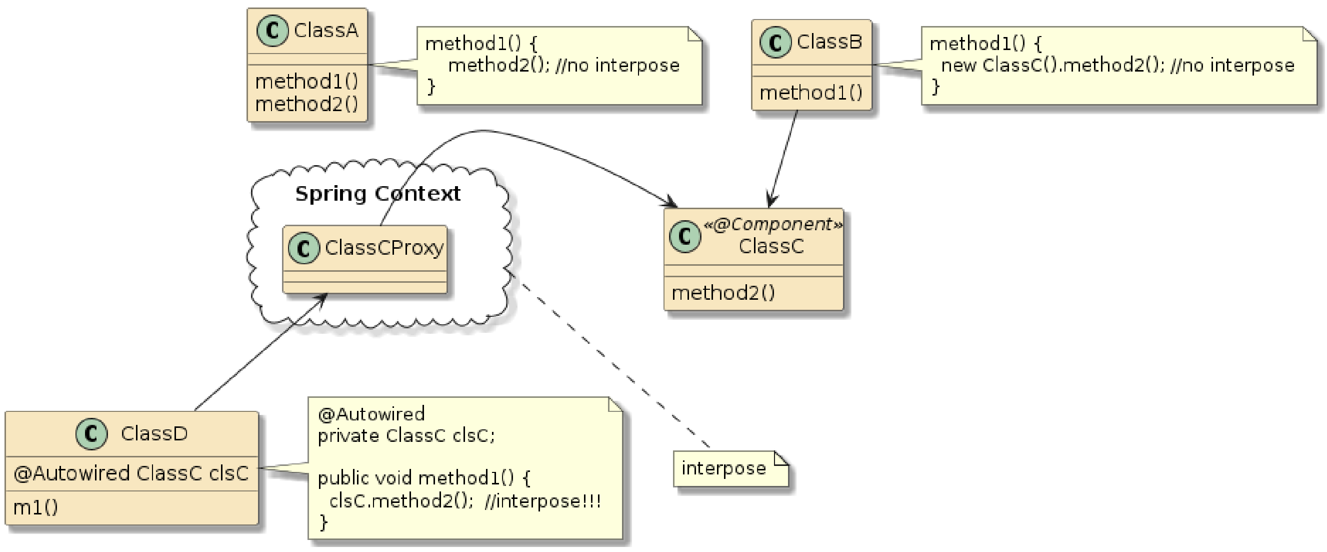


Figure 3. Interpose is only Automatic for Injected Components

1. **Buddy Method**: For the `ClassA` with `m1()` and `m2()` in the same class, Spring will normally not attempt to interpose a proxy in between those two methods (e.g., `@PreAuthorize`, `@Cacheable`). It is a straight Java call between methods of a class. That means no matter what annotations and constraints we define for `m2()` they will not be honored unless they are also on `m1()`. There is at least one exception for buddy methods, for `@Configuration(proxyBeanMethods=true)` — where a CGLIB proxy class will intercept calls between `@Bean` methods to prevent direct buddy method calls from instantiating independent POJO instances per call (i.e., not singleton components).

2. **Manually Instantiated**: For `ClassB` where `m2()` has been moved to a separate class but manually instantiated — no interpose takes place. This is a straight Java call between methods of two different classes. That also means that no matter what annotations are defined for `m2()`, they will not be honored unless they are also in place on `m1()`. It does not matter that `ClassC` is annotated as a `@Component` since `ClassB.m1()` manually instantiated it versus obtaining it from the Spring Context.

3. **Injected**: For `ClassD`, an instance of `ClassC` is injected. That means that the injected object has a chance to be a proxy class (either JDK Dynamic Proxy or CGLIB Proxy) to enforce the constraints defined on `ClassC.m2()`.

Keep this in mind as you work with various Spring configurations and review the following sections.

Chapter 7. Spring AOP

Spring Aspect Oriented Programming (AOP) provides a framework where we can define cross-cutting behavior to injected `@Components` using one or more of the available proxy capabilities behind the scenes. Spring AOP Proxy uses JDK Dynamic Proxy to proxy beans with interfaces and CGLIB to proxy bean classes lacking interfaces.

Spring AOP is a very capable but scaled back and simplified implementation of AspectJ. All the capabilities of AspectJ are allowed within Spring. However, the features integrated into Spring AOP itself are limited to method proxies formed at runtime. The compile-time byte manipulation offered by AspectJ is not part of Spring AOP.

7.1. AOP Definitions

The following represent some core definitions to AOP. Advice, AOP proxy, target object and (conceptually) the join point should look familiar to you. The biggest new concept here is the pointcut predicate that is used to locate the join point and how that is all modularized through a concept called aspect.

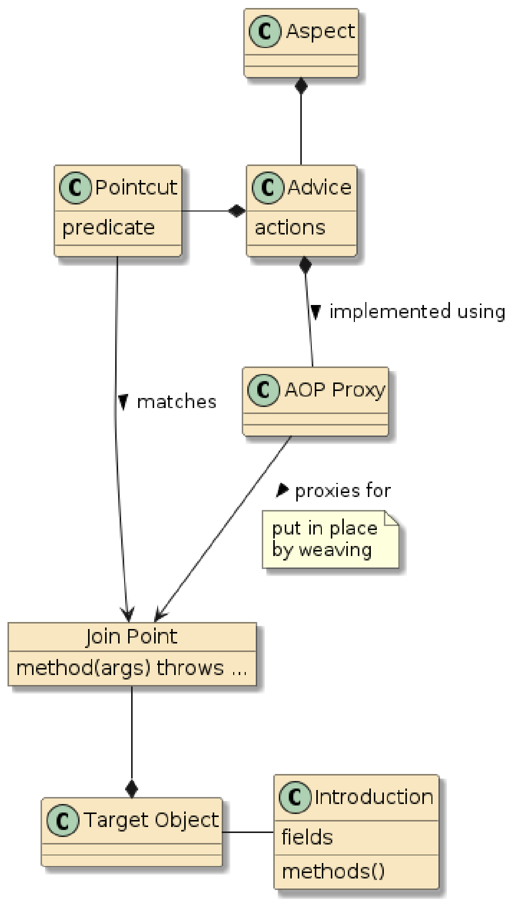


Figure 4. AOP Key Terms

Join Point is a point in the program (e.g., calling a method or throwing exception) in which we want to inject some code. For Spring AOP — this is always an event related to a method. AspectJ offers more types of join points.

Pointcut is a predicate rule that matches against a join point (i.e., a method begin, success, exception, or finally) and associates advice (i.e., more code) to execute at that point in the program. Spring uses the AspectJ pointcut language.

Advice is an action to be taken at a join point. This can be before, after (success, exception, or always), or around a method call. Advice chains are formed much the same as Filter chains of the web tier.

AOP proxy is an object created by AOP framework to implement advice against join points that match the pointcut predicate rule.

Aspect is a modularization of a concern that cuts across multiple classes/methods (e.g., timing measurement, security auditing, transaction boundaries). An aspect is made up of one or more advice action(s) with an assigned pointcut predicate.

Target object is an object being advised by one or more aspects. Spring uses proxies to implement advised (target) objects.

Introduction is declaring additional methods or fields on behalf of a type for an advised object, allowing us to add an additional interface and implementation.

Weaving is the linking aspects to objects. Spring AOP does this at runtime. AspectJ offers compile-time capabilities.

7.2. Enabling Spring AOP

To use Spring AOP, we must first add a dependency on `spring-boot-starter-aop`. That adds a dependency on `spring-aop` and `aspectj-weaver`.

Spring AOP Maven Dependency

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

We enable Spring AOP within our Spring Boot application by adding the `@EnableAspectJProxy` annotation to a `@Configuration` class or to the `@SpringBootApplication` class.

Enabling Spring AOP using Annotation

```
...
import org.springframework.context.annotation.EnableAspectJAutoProxy;
...
@Configuration
@EnableAspectJAutoProxy
public class ...
```

7.3. Aspect Class

Starting at the top — we have the `Aspect` class. This is a special `@Component` that defines the pointcut predicates to match and advice (before, after success, after throws, after finally, and around) to execute for join points.

Example Aspect Class

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

@Component ①
@Aspect ②
public class ItemsAspect {
    //pointcuts
    //advice
}
```

① annotated `@Component` to be processed by the application context

② annotated as `@Aspect` to have pointcuts and advice inspected

7.4. Pointcut

In Spring AOP — a pointcut is a predicate rule that identifies the method join points to match against for Spring beans (only). To help reduce complexity of definition, when using annotations — pointcut predicate rules are expressed in two parts:

- pointcut expression that determines exactly which method executions we are interested in
- signature with name and parameters

The signature is a method that returns void. The method name and parameters will be usable in later advice declarations. Although, the abstract example below does not show any parameters, they will become quite useful when we begin injecting typed parameters.

Example Pointcut

```
import org.aspectj.lang.annotation.Pointcut;
...
@Pointcut(/* pointcut expression*/) ①
public void serviceMethod(/* pointcut parameters */) {} //pointcut signature ②
```

① pointcut expression defines predicate matching rule(s)

② pointcut signature defines a name and parameter types for the pointcut expression

7.5. Pointcut Expression

The Spring AOP pointcut expressions use the the AspectJ pointcut language. Supporting the following designators

• **execution**	match method execution join points
• **within**	match methods below a package or type
• **@within**	match methods of a type that has been annotated with a given annotation
• **this**	match the proxy for a given type — useful when injecting typed advice arguments
• **target**	match the target for a given type — useful when injecting typed advice arguments
• **@target**	match methods of a type that has been annotated with specific annotation
• **@annotation**	match methods that have been annotated with a given annotation
• **args**	match methods that accept arguments matching this criteria
• **@args**	match methods that accept arguments annotated with a given annotation

• **bean**	Spring AOP extension to match Spring bean(s) based on a name or wildcard name expression

Don't use pointcut contextual designators for matching

Spring AOP Documentation recommends we use `within` and/or `execution` as our first choice of performant predicate matching and add contextual designators (`args`, `@annotation`, `this`, `target`, etc.) when needed for additional work versus using contextual designators alone for matching.

7.6. Example Pointcut Definition

The following example will match against any method in the services package, taking any number of arguments and returning any return type.

Example execution Pointcut

```
//execution(<return type> <package>.<class>.<method>(params))
@Pointcut("execution(* info.ejava.examples.svc.aop.items.services.*.*(..))")
//expression
public void serviceMethod() {} //signature
```

7.7. Combining Pointcut Expressions

We can combine pointcut definitions into compound definitions by referencing them and joining with a boolean ("&&" or "||") expression. The example below adds an additional condition to `serviceMethod()` that restricts matches to methods accepting a single parameter of type `GrillDTO`.

Example Combining Pointcut Expressions

```
@Pointcut("args(info.ejava.examples.svc.aop.items.dto.GrillDTO)") //expression
public void grillArgs() {} //signature

@Pointcut("serviceMethod() && grillArgs()") //expression
public void serviceMethodWithGrillArgs() {} //signature
```

7.8. Advice

The code that will act on the join point is specified in a method of the `@Aspect` class and annotated with one of the advice annotations. The following is an example of advice that executes before a join point.

Example Advice

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

@Component
@Aspect
@Slf4j
public class ItemsAspect {
    ...
    @Before("serviceMethodWithGrillArgs()")
    public void beforeGrillServiceMethod() {
        log.info("beforeGrillServiceMethod");
    }
```

The following table contains a list of the available advice types:

Table 1. Available Advice Types

@Before	runs prior to calling join point
@AfterReturning	runs after successful return from join point
@AfterThrowing	runs after exception from join point
@After	runs after join point no matter — i.e., finally
@Around	runs around join point. Advice must call join point and return result.

An example of each is towards the end of these lecture notes. For now, lets go into detail on some of the things we have covered.

Chapter 8. Pointcut Expression Examples

Pointcut expressions can be very expressive and can take some time to fully understand. The following examples should provide a head start in understanding the purpose of each and how they can be used. Other examples are available in the Spring AOP page.

8.1. execution Pointcut Expression

The execution expression allows for the definition of several pattern elements that can identify the point of a method call. The full format is as follows. [1]

execution Pointcut Expression Elements

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
pattern(param-pattern) throws-pattern?)
```

However, only the return type, name, and parameter definitions are required.

Required execution Patterns

```
execution(ret-type-pattern name-pattern(param-pattern))
```

The specific patterns include:

- **modifiers-pattern** - OPTIONAL access definition (e.g., public, protected)
- **ret-type-pattern** - MANDATORY type pattern for return type

  Example Return Type Patterns

  ```
  execution(info.ejava.examples.svc.aop.items.dto.GrillDTO *(..)) ①
  execution(*..GrillDTO *(..)) ②
  ```

  ① matches methods that return an explicit type

  ② matches methods that return `GrillDTO` type from any package

- **declaring-type-pattern** - OPTIONAL type pattern for package and class

  Example Declaring Type (package and class) Pattern

  ```
  execution(* info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.*(..)) ①
  execution(* *..GrillsServiceImpl.*(..)) ②
  execution(* info.ejava.examples.svc..Grills*.*(..)) ③
  ```

  ① matches methods within an explicit class

  ② matches methods within a `GrillsServiceImpl` class from any package

  ③ matches methods from any class below `…svc` and start with letters `Grills`

* **name-pattern** - MANDATORY pattern for method name

  Example Name (method) Pattern

  ```
  execution(* createItem(..)) ①
  execution(* *..GrillsServiceImpl.createItem(..)) ②
  execution(* create*(..)) ③
  ```

  ① matches any method called `createItem` of any class of any package

  ② matches any method called `createItem` within class `GrillsServiceImpl` of any package

  ③ matches any method of any class of any package that starts with the letters `create`

* **param-pattern** - MANDATORY pattern to match method arguments. `()` will match a method with no arguments. `(*)` will match a method with a single parameter. `(T,*)` will match a method with two parameters with the first parameter of type `T`. `(..)` will match a method with 0 or more parameters

  Example noargs () Pattern

  ```
  execution(void
  info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.deleteItems())①
  execution(* *..GrillsServiceImpl.*()) ②
  execution(* *..GrillsServiceImpl.delete*()) ③
  ```

  ① matches an explicit method that takes no arguments

  ② matches any method within a `GrillsServiceImpl` class of any package and takes no arguments

  ③ matches any method from the `GrillsServiceImpl` class of any package, taking no arguments, and the method name starts with `delete`

  Example Single Argument Patterns

  ```
  execution(*
  info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.createItem(*))①
  execution(* createItem(info.ejava.examples.svc.aop.items.dto.GrillDTO)) ②
  execution(* *(*..GrillDTO)) ③
  ```

  ① matches an explicit method that accepts any single argument

  ② matches any method called `createItem` that accepts a single parameter of a specific type

  ③ matches any method that accepts a single parameter of `GrillDTO` from any package

  Example Multiple Argument Patterns

  ```
  execution(*
  info.ejava.examples.svc.aop.items.services.GrillsServiceImpl.updateItem(*,*))①
  execution(* updateItem(int,*)) ②
  execution(* updateItem(int,*..GrillDTO)) ③
  ```

① matches an explicit method that accepts two arguments of any type

② matches any method called `updateItem` that accepts two arguments of type `int` and any second type

③ matches any method called `updateItem` that accepts two arguments of type `int` and `GrillDTO` from any package

8.2. within Pointcut Expression

The within pointcut expression is similar to supplying an `execution` expression with just the declaring type pattern specified.

Example within Expressions

```
within(info.ejava.examples.svc.aop.items..*) ①
within(*..ItemsService+) ②
within(*..BedsServiceImpl) ③
```

① match all methods in package `info.ejava.examples.svc.aop.items` and its subpackages

② match all methods in classes that implement `ItemsService` interface

③ match all methods in `BedsServiceImpl` class

8.3. target and this Pointcut Expressions

The `target` and `this` pointcut designators are very close in concept to `within` when used in the following way. The difference will show up when we later use them to inject typed arguments into the advice. These are considered "contextual" designators and are primarily placed in the predicate to pull out members of the call for injection.

```
target(info.ejava.examples.svc.aop.items.services.BedsServiceImpl) ①
this(info.ejava.examples.svc.aop.items.services.BedsServiceImpl) ②
```

① matches methods of target object — object being proxied — is of type

② matches methods of proxy object — object implementing proxy — is of type

```
@target(org.springframework.stereotype.Service) ①
@annotation(org.springframework.core.annotation.Order) ②
```

① matches all methods in class annotated with `@Service`

② matches all methods having annotation `@Order`

[1] *"Spring AOP execution Examples"*, Spring AOP

Chapter 9. Advice Parameters

Our advice methods can accept two types of parameters:

- typed using context designators
- dynamic using `JoinPoint`

Context designators like `args`, `@annotation`, `target`, and `this` allow us to assign a logical name to a specific part of a method call so that can be injected into our advice method.

Dynamic injection involves a single `JointPoint` object that can answer the contextual details of the call.

> *Do not use context designators alone as predicates to locate join points*
>
> The Spring AOP documentation recommends using `within` and `execution` designators to identify a pointcut and contextual designators like `args` to bind aspects of the call to input parameters. That is guidance is not fully followed in the following context examples. We easily could have made the non-contextual designators more explicit.

9.1. Typed Advice Parameters

We can use the `args` expression in the pointcut to identify criteria for parameters to the method and to specifically access one or more of them.

The left side of the following pointcut expression matches on all executions of methods called `createGrill()` taking any number of arguments. The right side of the pointcut expression matches on methods with a single argument. When we match that with the `createGrill` signature — the single argument must be of the type `GrillDTO`

Example Single, Typed Argument

```
@Pointcut("execution(* createItem(..)) && args(grillDTO)") ① ②
public void createGrill(GrillDTO grillDTO) {} ③

@Before("createGrill(grill)") ④
public void beforeCreateGrillAdvice(GrillDTO grill) { ⑤
    log.info("beforeCreateGrillAdvice: {}", grill);
}
```

① left hand side of pointcut expression matches execution of `createItem` methods with any parameters

② right hand side of pointcut expression matches methods with a single argument and maps that to name `grillDTO`

③ pointcut signature maps `grillDTO` to a Java type — the names within the pointcut must match

④ advice expression references `createGrill` pointcut and maps first parameter to name `grill`

⑤ advice method signature maps name `grill` to a Java type — the names within the advice must match but do not need to match the names of the pointcut

The following is logged before the createGrill method is called.

Example Single, Typed Argument Output

```
beforeCreateGrillAdvice: GrillDTO(super=ItemDTO(id=0, name=weber))
```

9.2. Multiple,Typed Advice Parameters

We can use the `args` designator to specify multiple arguments as well. The right hand side of the pointcut expression matches methods that accept two parameters. The pointcut method signature maps these to parameters to Java types. The example advice references the pointcut but happens to use different parameter names. The names used match the parameters used in the advice method signature.

Example Multiple, Typed Arguments

```
@Pointcut("execution(* updateItem(..)) && args(grillId, updatedGrill)")
public void updateGrill(int grillId, GrillDTO updatedGrill) {}

@Before("updateGrill(id, grill)")
public void beforeUpdateGrillAdvice(int id, GrillDTO grill) {
    log.info("beforeUpdateGrillAdvice: {}, {}", id, grill);
}
```

The following is logged before the updateGrill method is called.

Example Multiple, Typed Arguments Output

```
beforeUpdateGrillAdvice: 1, GrillDTO(super=ItemDTO(id=0, name=green egg)
```

9.3. Annotation Parameters

We can target annotated classes and methods and make the value of the annotation available to the advice using the pointcut signature mapping. In the example below, we want to match on all methods below the `items` package that have an `@Order` annotation and pass that annotation as a parameter to the advice.

Example @Annotation Parameter

```
import org.springframework.core.annotation.Order;
...
@Pointcut("@annotation(order)") ①
public void orderAnnotationValue(Order order) {} ②

@Before("within(info.ejava.examples.svc.aop.items..*) && orderAnnotationValue(order)")
```

```
public void beforeOrderAnnotation(Order order) { ③
    log.info("before@OrderAnnotation: order={}", order.value()); ④
}
```

① we are targeting methods with an annotation and mapping that to the name `order`

② the name `order` is being mapped to the type `org.springframework.core.annotation.Order`

③ the `@Order` annotation instance is being passed into advice

④ the value for the `@Order` annotation can be accessed

I have annotated one of the candidate methods with the `@Order` annotation and assigned a value of `100`.

Example @Annotation Parameter Target Method

```
import org.springframework.core.annotation.Order;
...
@Service
public class BedsServiceImpl extends ItemsServiceImpl<BedDTO> {
    @Override
    @Order(100)
    public BedDTO createItem(BedDTO item) {
```

In the output below — we see that the annotation was passed into the advice and provided with the value `100`.

Example @Annotation Parameter Output

```
before@OrderAnnotation: order=100
```

Annotations can pass contextual values to advice

Think how a feature like this — where an annotation on a method with attribute values — can be of use with security role annotations.

9.4. Target and Proxy Parameters

We can map the target and proxy references into the advice method using the `target()` and `this()` designators. In the example below, the `target` name is mapped to the `ItemsService<BedsDTO>` interface and the `proxy` name is mapped to a vanilla `java.lang.Object`. The `target` type mapping constrains this to calls to the `BedsServiceImpl`.

Example target and this Parameters

```
@Before("target(target) && this(proxy)")
public void beforeTarget(ItemsService<BedDTO> target, Object proxy) {
    log.info("beforeTarget: target={}, proxy={}",target.getClass(),proxy.getClass());
}
```

The advice prints the name of each class. The output below shows that the target is of the target implementation type (i.e., no proxy layer) and the proxy is of a CGLIB proxy type (i.e., it is the proxy to the target).

Example target and this Parameters Result

```
beforeTarget:
  target=class info.ejava.examples.svc.aop.items.services.BedsServiceImpl,
  proxy=class
info.ejava.examples.svc.aop.items.services.BedsServiceImpl$$EnhancerBySpringCGLIB$$a38
982b5
```

9.5. Dynamic Parameters

If we have generic pointcuts and do not know ahead of time which parameters we will get and in what order, we can inject a `JoinPoint` parameter as the first argument to the advice. This object has many methods that provide dynamic access to the context of the method — including parameters. The example below logs the classname, method, and array of parameters in the call.

Example JointPoint Injection

```
@Before("execution(* *..Grills*.*(..))")
public void beforeGrillsMethodsUnknown(JoinPoint jp) {
    log.info("beforeGrillsMethodsUnknown: {}.{}, {}",
            jp.getTarget().getClass().getSimpleName(),
            jp.getSignature().getName(),
            jp.getArgs());
}
```

9.6. Dynamic Parameters Output

The following output shows two sets of calls: `createItem` and `updateItem`. Each were intercepted at the controller and service level.

Example JointPoint Injection Output

```
beforeGrillsMethodsUnknown: GrillsController.createItem,
                                    [GrillDTO(super=ItemDTO(id=0, name=weber))]
beforeGrillsMethodsUnknown: GrillsServiceImpl.createItem,
                                    [GrillDTO(super=ItemDTO(id=0, name=weber))]
beforeGrillsMethodsUnknown: GrillsController.updateItem,
                                    [1, GrillDTO(super=ItemDTO(id=0, name=green egg))]
beforeGrillsMethodsUnknown: GrillsServiceImpl.updateItem,
                                    [1, GrillDTO(super=ItemDTO(id=0, name=green egg))]
```

Chapter 10. Advice Types

We have five advice types:

- @Before
- @AfterReturning
- @AfterThrowing
- @After
- @Around

For the first four — using `JoinPoint` is optional. The last type (`@Around`) is required to inject `ProceedingJoinPoint` — a subclass of `JoinPoint` — in order to delegate to the target and handle the result. Lets take a look at each in order to have a complete set of examples.

To demonstrate, I am going to define an advice of each type that will use the same pointcut below.

Example Pointcut to Demonstrate Advice Types

```
@Pointcut("execution(* *..MowersServiceImpl.updateItem(*,*)) && args(id,mowerUpdate)"
)①
public void mowerUpdate(int id, MowerDTO mowerUpdate) {} ②
```

① matches all `updateItem` methods calls in the `MowersServiceImpl` class taking two arguments

② arguments will be mapped to type `int` and `MowerDTO`

There will be two matching calls:

1. the first will be successful
2. the second will throw a NotFound RuntimeException.

10.1. @Before

The Before advice will be called prior to invoking the join point method. It has access to the input parameters and can change the contents of them. This advice does not have access to the result.

Example @Before Advice

```
@Before("mowerUpdate(id, mowerUpdate)")
public void beforeMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate) {
    log.info("beforeMowerUpdate: {}, {}", id, mowerUpdate);
}
```

The before advice only has access to the input parameters prior to making the call. It can modify the parameters, but not swap them around. It has no insight into what the result will be.

@Before Advice Example for Successful Call

```
beforeMowerUpdate: 1, MowerDTO(super=ItemDTO(id=0, name=bush hog))
```

Since the before advice is called prior to the join point, it is oblivious that this call ended in an exception.

@Before Advice Example for Call Throwing Exception

```
beforeMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
```

10.2. @AfterReturning

After returning advice will get called when a join point successfully returns without throwing an exception. We have access to the result through an annotation field and can map that to an input parameter.

Example @AfterReturning Advice

```
@AfterReturning(value = "mowerUpdate(id, mowerUpdate)",
    returning = "result")
public void afterReturningMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate,
MowerDTO result) {
    log.info("afterReturningMowerUpdate: {}, {} => {}", id, mowerUpdate, result);
}
```

The `@AfterReturning` advice is called only after the successful call and not the exception case. We have access to the input parameters and the result. The result can be changed before returning to the caller. However, the input parameters have already been processed.

@AfterReturning Advice Example for Successful Call

```
afterReturningMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))
    => MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

10.3. @AfterThrowing

The `@AfterThrowing` advice is called only when an exception is thrown. Like the successful sibling, we can map the resulting exception to an input variable to make it accessible to the advice.

Example @AfterThrowing Advice

```
@AfterThrowing(value = "mowerUpdate(id, mowerUpdate)", throwing = "ex")
public void afterThrowingMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate,
ClientErrorException.NotFoundException ex) {
    log.info("afterThrowingMowerUpdate: {}, {} => {}", id,mowerUpdate,ex.toString());
}
```

The `@AfterThrowing` advice has access to the input parameters and the exception. The exception will still be thrown after the advice is complete. I am not aware of any ability to squelch the exception and return a non-exception here. Look to `@Around` to give you that capability at a minimum.

@AfterThrowing Advice Example for Call Throwing Exception

```
afterThrowingMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
    => info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:
item[2] not found
```

10.4. @After

`@After` is called after a successful return or exception thrown. It represents logic that would commonly appear in a `finally` block to close out resources.

Example @After Advice

```
@After("mowerUpdate(id, mowerUpdate)")
public void afterMowerUpdate(JoinPoint jp, int id, MowerDTO mowerUpdate) {
    log.info("afterReturningMowerUpdate: {}, {}", id, mowerUpdate);
}
```

The `@After` advice is always called once the joint point finishes executing.

@After Advice Example for Successful Call

```
afterReturningMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

@After Advice Example for Call Throwing Exception

```
afterReturningMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
```

10.5. @Around

`@Around` is the most capable advice but possibly the more expensive one to execute. It has full control over the input and return values and whether the call is made at all. The example below logs the various paths through the advice.

Example @Around Advice

```
@Around("mowerUpdate(id, mowerUpdate)")
public Object aroundMowerUpdate(ProceedingJoinPoint pjp, int id, MowerDTO mowerUpdate)
throws Throwable {
    Object result = null;
    try {
        log.info("entering aroundMowerUpdate: {}, {}", id, mowerUpdate);
        result = pjp.proceed(pjp.getArgs());
        log.info("returning after successful aroundMowerUpdate: {}, {} => {}", id,
mowerUpdate, result);
        return result;
    } catch (Throwable ex) {
        log.info("returning after aroundMowerUpdate excdeption: {}, {} => {}", id,
mowerUpdate, ex.toString());
        result = ex;
        throw ex;
    } finally {
        log.info("returning after aroundMowerUpdate: {}, {} => {}",
                id, mowerUpdate, (result==null ? null :result.toString()));
    }
}
```

The `@Around` advice example will log activity prior to calling the join point, after successful return from join point, and finally after all advice complete.

@Around Advice Example for Successful Call

```
entering aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=0, name=bush hog))
returning after successful aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1,
name=bush hog))
    => MowerDTO(super=ItemDTO(id=1, name=bush hog))
returning after aroundMowerUpdate: 1, MowerDTO(super=ItemDTO(id=1, name=bush hog))
    => MowerDTO(super=ItemDTO(id=1, name=bush hog))
```

The `@Around` advice example will log activity prior to calling the join point, after an exception from the join point, and finally after all advice complete.

@Around Advice Example for Call Throwing Exception

```
entering aroundMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
returning after aroundMowerUpdate exception: 2, MowerDTO(super=ItemDTO(id=0, name=john
deer))
    => info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:
item[2] not found
returning after aroundMowerUpdate: 2, MowerDTO(super=ItemDTO(id=0, name=john deer))
    => info.ejava.examples.common.exceptions.ClientErrorException$NotFoundException:
item[2] not found
```

Chapter 11. Other Features

We have covered a lot of capability in this chapter and likely all you will need. However, know there were a few other topics left unaddressed that I thought might be of interest in certain circumstances.

- Ordering - useful when we declare multiple advice for the same join point and need one to run before the other
- Introductions - a way to add additional state and behavior to a join point/target instance
- Programmatic Spring AOP proxy creation - a way to create Spring AOP proxies on the fly versus relying on injection. This is useful for data value objects that are typically manually created to represent a certain piece of information.
- Schema Based AOP Definitions - Spring also offers an means to express AOP behavior using XML. They are very close in capability — so if you need the ability to flexibly edit aspects in production without changing the Java code — this is an attractive option.

Chapter 12. Summary

In this module we learned:

- how we can decouple potentially cross-cutting logic from business code using different levels of dynamic invocation technology
- to obtain and invoke a method reference using Java Reflection
- to encapsulate advice within proxy classes using interfaces and JDK Dynamic Proxies
- to encapsulate advice within proxy classes using classes and CGLIB dynamically written sub-classes
- to integrate Spring AOP into our project
- to identify method join points using AspectJ language
- to implement different types of advice (before, after (completion, exception, finally), and around)
- to inject contextual objects as parameters to advice

After learning this material you will surely be able to automatically envision the implementation techniques used by Spring in order to add framework capabilities to our custom business objects. Those interfaces we implement and annotations we assign are likely the target of many Spring AOP aspects, adding advice in a configurable way.