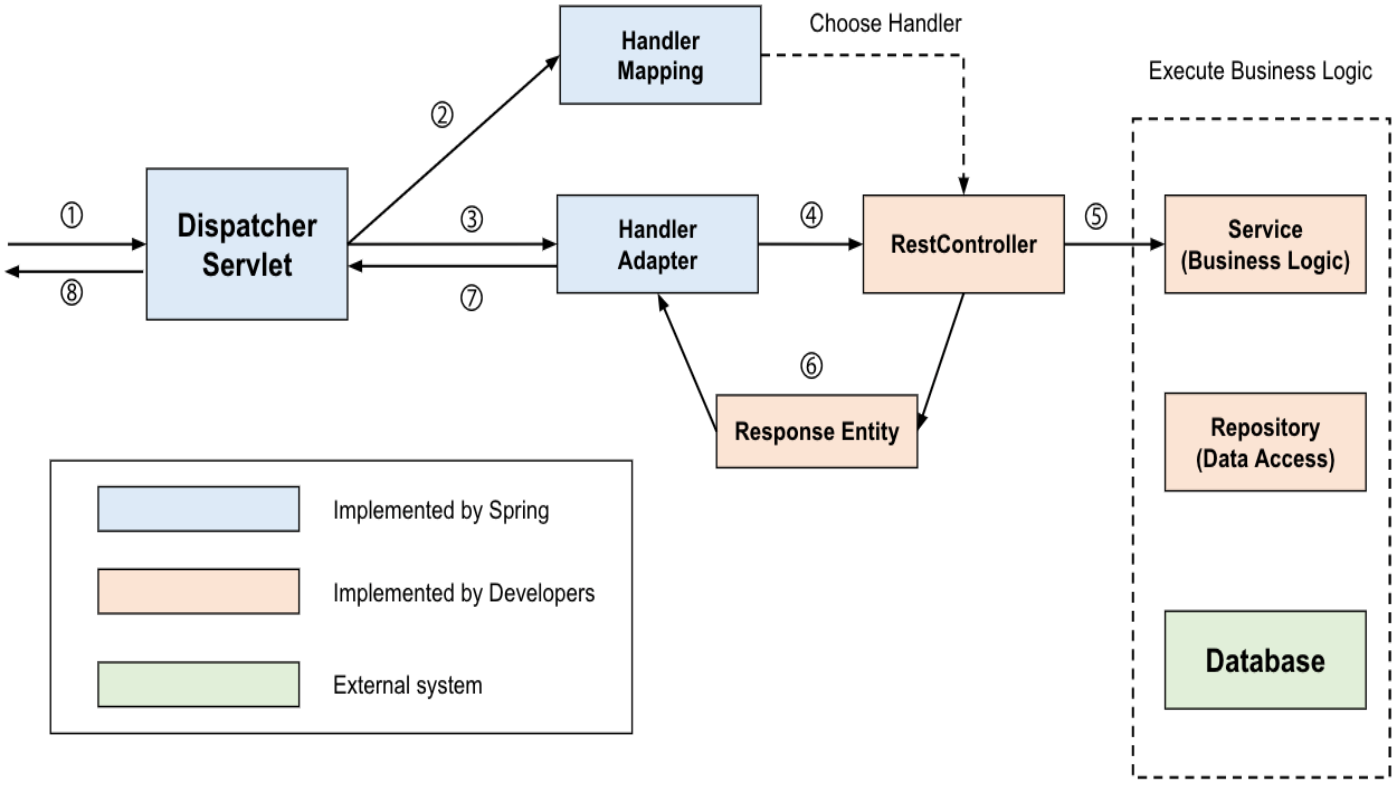
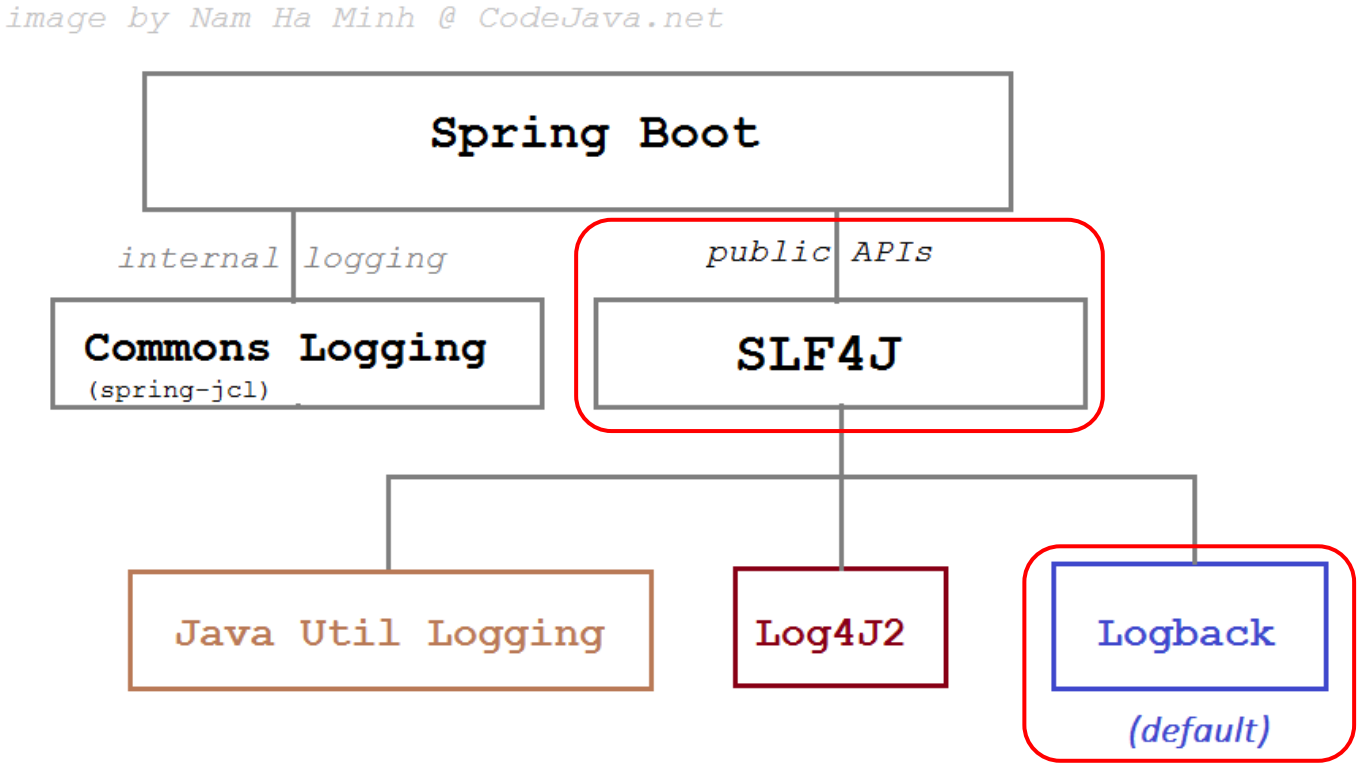
[DispatcherServlet 의 동작 과정]



스프링 부트 로깅

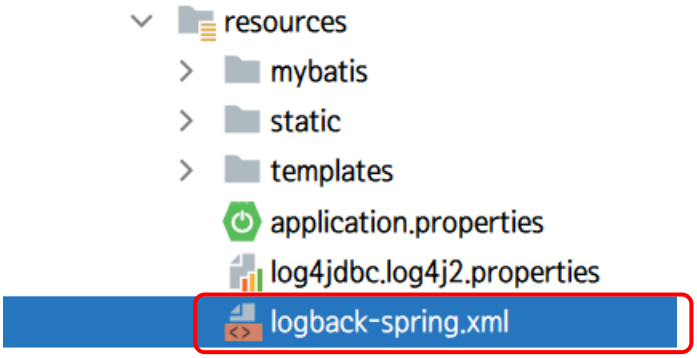
[스프링의 기본 로깅 구조]



LogBack 은 가장 많이 사용되었던 로깅 라이브러리인 Log4j 의 후속 버전으로 Log4j 를 더욱 성능을 좋게 만든 Java 의 오픈 소스 Logging Framework 이다.
스프링 부트에 기본으로 설정 되어 있어 별도의 라이브러리를 추가하지 않아도 된다.

(Log4J 의 보안 취약점 : https://www.lgcns.com/blog/cns-tech/security/22605/77)

2. 설정 방법
1) resources 에 'logback-spring.xml' 파일을 생성한다.



spring-boot-starter-web 안에 spring-boot-starter-logging 에 구현체가 있다.

LogBack 을 이용하여 로깅을 수행하기 위해서 필요한 주요 설정요소로는 Logger, Appender, Encoder 등 3 가지가 있으며 각각은 다음과 같은 역할을 수행한다.

TRACE < DEBUG < INFO < WARN < ERROR

1) ERROR : 요청을 처리하는 중 오류가 발생한 경우 표시한다.
2) WARN : 처리 가능한 문제, 향후 시스템 에러의 원인이 될 수 있는 경고성 메시지를 나타낸다.
3) INFO : 상태변경과 같은 정보성 로그를 표시한다.
4) DEBUG : 프로그램을 디버깅하기 위한 정보를 표시한다.
5) TRACE : Debug 보다 훨씬 상세한 정보를 표시한다. (처리 흐름을 추적하기 위한)

로그에 설정할 수 있는 레벨은 총 5 가지다. 출력 레벨의 설정에 따라 설정 레벨 이상의 로그를 출력한다. 로깅 레벨 설정을 "INFO"로 하였을 경우 "TRACE", "DEBUG" 레벨은 무시한다.
스프링 부트에서는 간단히 application.properties 에 값만 세팅해도 설정 가능 하다.

```
# 루트 레벨(전체 레벨) 전체 로깅 레벨 지정  --> 디폴트
logging.level.root=info
```

```
# 패키지별로 로깅 레벨 지정
```

다음 방법으로 상위 패키지의 디폴트 레벨을 설정하고, 하위 패키지들에 대한 각각의 로깅 레벨을 별도로 설정할 수 있다.

```
logging.level.org.springframework.web=info
logging.level.com.example.springedu=debug
logging.level.thymeleaf.exam=trace
```

[logback-spring.xml 설정]

- 대소문자 구별하지 않는다.
- name attribute 를 반드시 지정해야 한다.
- logback-spring.xml 은 appender 와 logger 크게 두개로 구분된다.
- Dynamic Reloading 기능을 지원한다.
  ex) 60 초 주기마다 로그파일(logback-spring.xml)이 바뀌었는지 검사하고 바뀌었으면 프로그램 갱신한다.

1) appender
   로그 메시지가 출력될 대상을 결정하는 요소이다.
   (콘솔에 출력할지, 파일로 출력 할지 등의 설정)

- appender 의 class 의 종류

1) ch.qos.logback.core.ConsoleAppender
   콘솔에 로그를 찍음, 로그를 OutputStream 에 작성하여 콘솔에 출력되도록 한다.
2) ch.qos.logback.core.FileAppender
   파일에 로그를 찍음, 최대 보관 일 수 등을 지정할 수 있다.
3) ch.qos.logback.core.rolling.RollingFileAppender
   여러 개의 파일을 롤링, 순회하면서 로그를 찍는다.
   (지정 용량이 넘어간 Log File 을 넘버링 하여 나누거나 일별로 로그 파일을 생성하여 저장할 수 있다.)
4) ch.qos.logback.classic.net.SMTPAppender
   로그를 메일로 보낸다.
5) ch.qos.logback.classic.db.DBAppender
   DB(데이터베이스)에 로그를 저장한다.

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern>[%d{yyyy-MM-dd HH:mm:ss}][%thread] %-5level %logger{36} - %msg%n</pattern>
        </layout>
    </appender>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${LOGS_ABSOLUTE_PATH}/mylogback.log</file>
        <encoder>
            <pattern>[%d{yyyy-MM-dd HH:mm:ss}] %-5level %logger{35} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="ROLLINGFILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder>
            <pattern>[%d{yyyy-MM-dd HH:mm:ss}:%-3relative][%thread] %-5level %logger{35}
- %msg%n</pattern>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOGS_ABSOLUTE_PATH}/logback.%d{yyyy-MM-dd}.log</fileNamePattern>
        </rollingPolicy>
    </appender>
```

%logger	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar
%logger{0}	mainPackage.sub.sample.Bar	Bar
%logger{5}	mainPackage.sub.sample.Bar	m.s.s.Bar
%logger{10}	mainPackage.sub.sample.Bar	m.s.s.Bar
%logger{15}	mainPackage.sub.sample.Bar	m.s.sample.Bar
%logger{16}	mainPackage.sub.sample.Bar	m.sub.sample.Bar
%logger{26}	mainPackage.sub.sample.Bar	mainPackage.sub.sample.Bar

2) root(디폴트 logger)와 logger
package 와 level 을 설정하고 appender 를 참조하게 정의한다.

[root]
- 전역 로거 설정이다.
- 항상 마지막에 수행되는 기본 로거이다.

[logger]
- 지역 로거 설정이다.
- additivity 속성으로 root 설정을 마저 수행할 것인지의 여부를 결정 가능(default = true)

```
<root level="INFO">
    <appender-ref ref="STDOUT" />
</root>

<logger name="com.example.springedu.controller.LogTestController1" level="DEBUG">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="ROLLINGFILE" />
</logger>
<logger name="com.example.springedu.controller.LogTestController2" level="TRACE" additivity="false">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="FILE" />
</logger>
```

3) property
- 설정파일에서 사용될 변수값 선언

4) layout 과 encoder
- layout : 로그의 출력 포맷을 지정한다.
  참고 : http://logback.qos.ch/manual/layouts.html
- encoder : Appender 에 포함되며 출력될 로그메시지를 원하는 형식으로 변환하는 역할을 담당한다. FileAppender 에서는 encoder 를 사용하여 layout 은 설정한다.

5) file
기록할 파일명과 경로를 설정한다.

6) pattern

[패턴에 사용되는 요소]

%logger{length} : Logger name 을 축약할 수 있다. {length}는 최대 자리 수
%-5level : 로그 레벨, -5 는 출력의 고정폭 값(5 글자)

%msg : - 로그 메시지 (=%message)
${PID:-} : 프로세스 아이디
%d : 로그 기록시간
%p : 로깅 레벨
%F : 로깅이 발생한 프로그램 파일명
%M : 로깅일 발생한 메소드의 명
%l : 로깅이 발생한 호출지의 정보
%L : 로깅이 발생한 호출지의 라인 수
%thread : 현재 Thread 명
%t : 로깅이 발생한 Thread 명
%c : 로깅이 발생한 카테고리
%C : 로깅이 발생한 클래스 명
%m : 로그 메시지
%n : 줄바꿈(new line)
%% : %를 출력
%r : 애플리케이션 시작 이후부터 로깅이 발생한 시점까지의 시간(ms)

<file>

- rollingPolicy class
ch.qos.logback.core.rolling.TimeBasedRollingPolicy => 일자별 적용
ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP => 일자별 + 크기별 적용

- fileNamePattern
파일 쓰기가 종료된 log 파일명의 패턴을 지정한다.
.gz 나 .zip 으로 자동으로 압축할 수도 있다.

- maxFileSize
한 파일당 최대 파일 용량을 지정한다.
log 내용의 크기도 IO 성능에 영향을 미치기 때문에 되도록이면 너무 크지 않은 사이즈로 지정하는 것이 좋다.(최대 10MB 내외 권장)
용량의 단위는 KB, MB, GB 3 가지를 지정할 수 있다.
RollingFile 이름 패턴에 .gz 이나 .zip 을 입력할 경우 로그파일을 자동으로 압축해주는 기능도 지원한다.
- maxHistory
최대 파일 생성 갯수
예를들어 maxHistory 가 30 이고 Rolling 정책을 일 단위로 하면 30 일동안만 저장되고, 월 단위로 하면 30 개월간 저장된다.

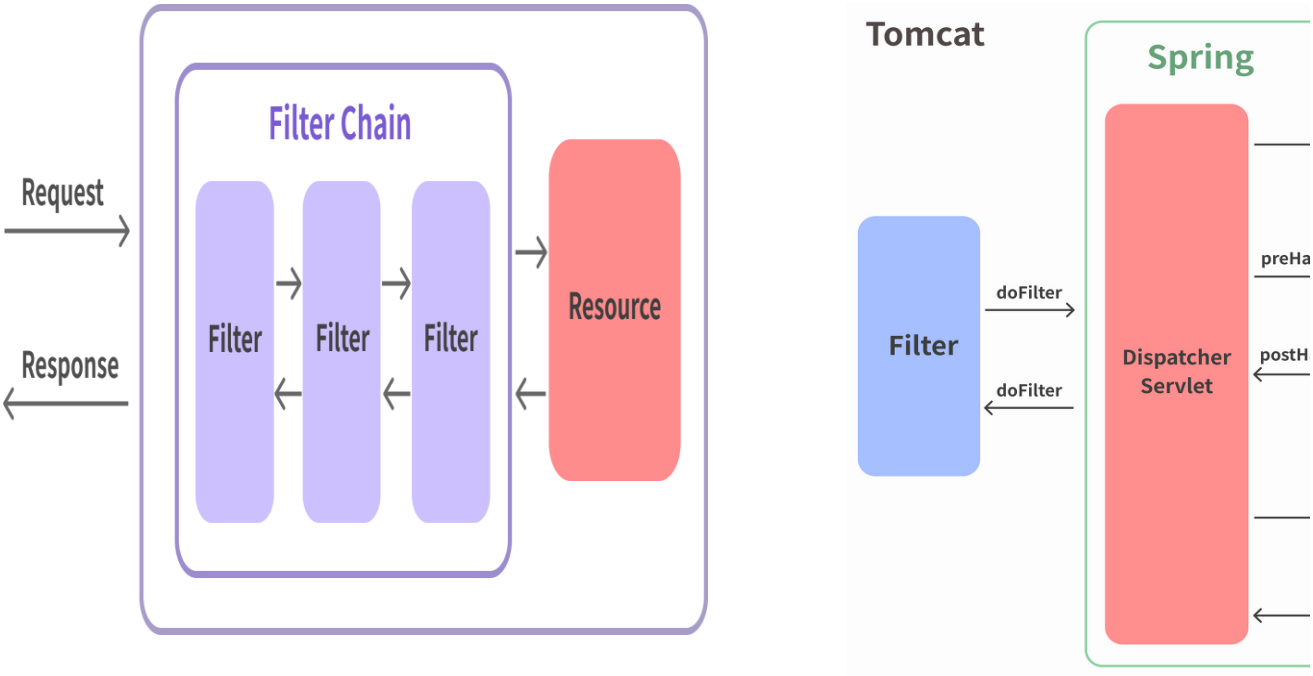
필터와 인터셉터

Spring은 공통적인 여러 작업을 대신 처리하며 개발시 중복된 코드를 제거할 수 있도록 많은 기능들을 지원하는데 필터(Filter)와 인터셉터(Interceptor)도 여기에 포함된다.

[필터(Filter)란?]

필터(Filter)는 J2EE 표준 스펙 기능으로 DispatcherServlet(Dispatcher Servlet)에 요청이 전달되기 전/후에 url 패턴에 맞는 모든 요청에 대해 부가작업을 처리할 수 있는 기능을 제공한다. DispatcherServlet은 스프링의 가장 앞단에 존재하는 프론트 컨트롤러이므로, 필터는 스프링 범위 밖에서 처리가 되는 것이다.

즉, 스프링 컨테이너가 아닌 톰캣과 같은 웹 컨테이너(서블릿 컨테이너)에 의해 관리가 되는 것이고(스프링 빈으로 등록은 된다), DispatcherServlet 전/후에 처리하는 것이다. 이러한 과정을 그림으로 표현하면 다음과 같다.



[필터(Filter)의 메서드]

필터를 추가하기 위해서는 javax.servlet의 Filter 인터페이스를 구현(implements)해야 하며 이는 다음의 3가지 메서드를 가지고 있다.

```
public interface Filter {
    public default void init(FilterConfig filterConfig) throws ServletException {}
    public void doFilter(ServletRequest request, ServletResponse response,
```

```
            FilterChain chain) throws IOException, ServletException {}
    public default void destroy() {}
}
```

- init 메서드

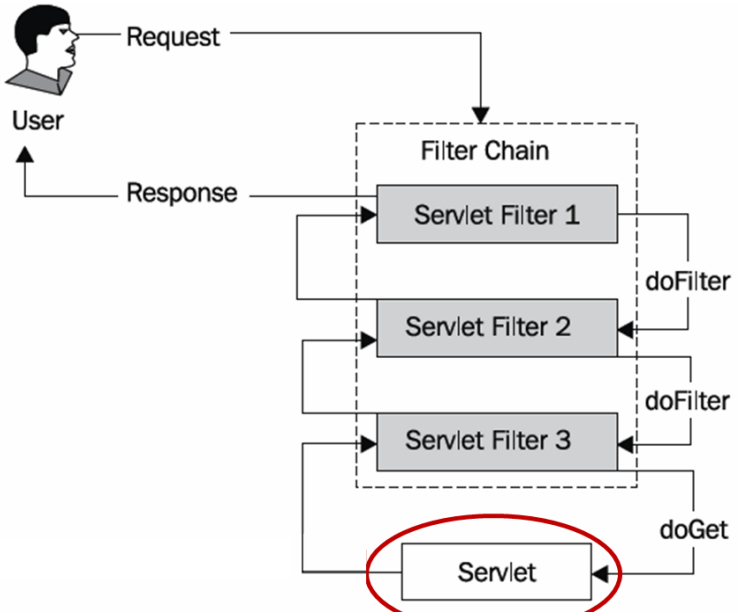
init 메서드는 필터 객체를 초기화하고 서비스에 추가하기 위한 메서드다. 웹 컨테이너가 1회 init 메서드를 호출하여 필터 객체를 초기화하면 이후의 요청들은 doFilter를 통해 처리된다.

- doFilter 메서드

doFilter 메서드는 url-pattern에 맞는 모든 HTTP 요청이 DispatcherServlet으로 전달되기 전에 웹 컨테이너에 의해 실행되는 메서드다. doFilter의 파라미터로는 FilterChain이 있는데, FilterChain의 doFilter 통해 다음 대상으로 요청을 전달하게 된다. chain.doFilter() 전/후에 우리가 필요한 처리 과정을 넣어줌으로써 원하는 처리를 진행할 수 있다.

- destroy 메서드

destroy 메서드는 필터 객체를 서비스에서 제거하고 사용하는 자원을 반환하기 위한 메서드다. 이는 웹 컨테이너에 의해 1번 호출되며 이후에는 이제 doFilter에 의해 처리되지 않는다.



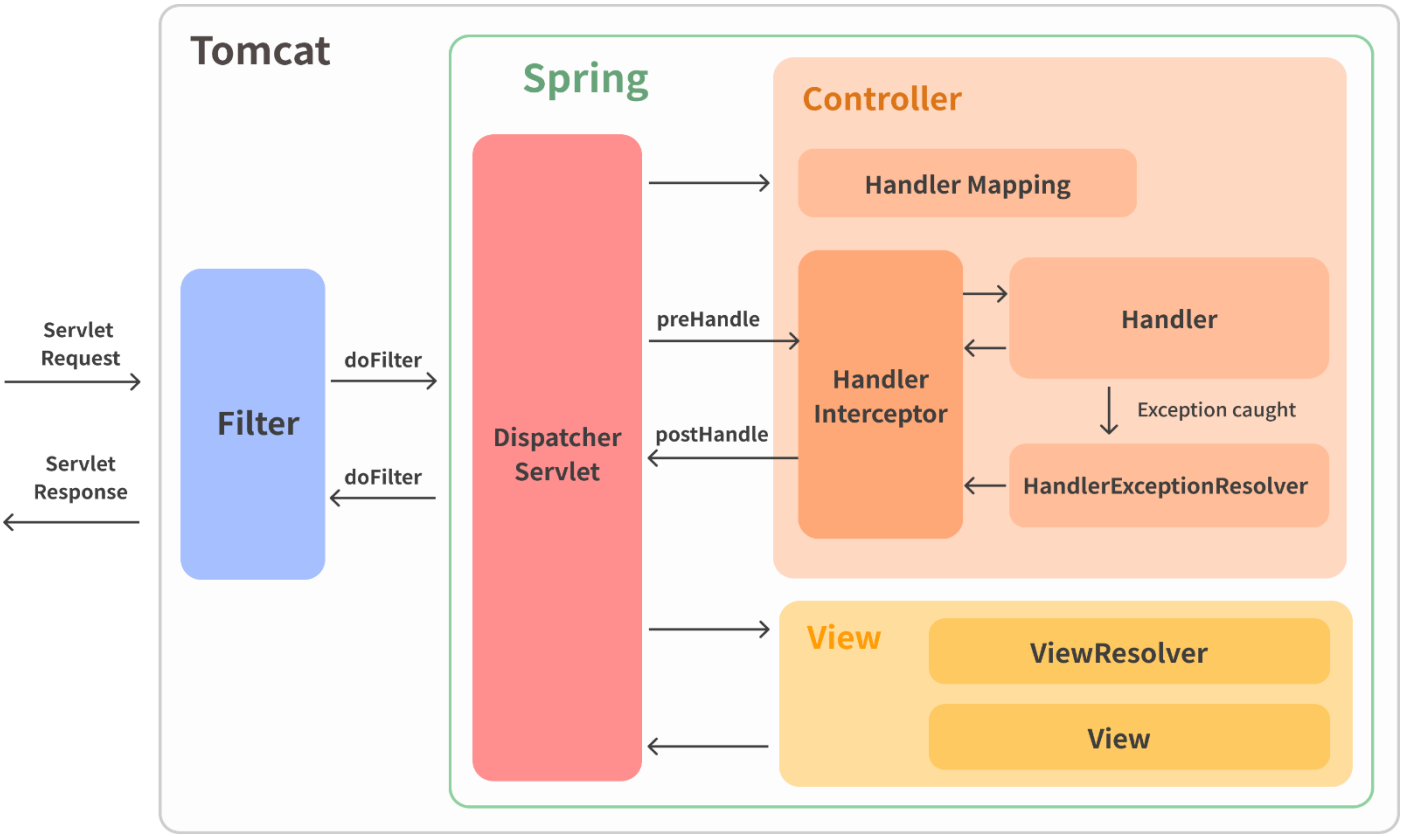
[필터의 등록]

```
@Component
@Slf4j
@Order(2)
public class TestFilter1 implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain)
```

```
                    throws IOException, ServletException{
        log.info("[필터1] 요청 자원 수행 전");
        chain.doFilter(request, response);
        log.info("[필터1] 요청 자원 수행 후");
    }
}
```

[인터셉터(Interceptor)란?]

인터셉터(Interceptor)는 J2EE 표준 스펙인 필터(Filter)와 달리 Spring이 제공하는 기술로써, DispatcherServlet(Dispatcher Servlet)이 컨트롤러를 호출하기 전과 후에 요청과 응답을 참조하거나 가공할 수 있는 기능을 제공한다. 즉, 웹 컨테이너(서블릿 컨테이너)에서 동작하는 필터와 달리 인터셉터는 스프링 컨텍스트에서 동작을 하는 것이다.
DispatcherServlet은 핸들러 매핑을 통해 적절한 컨트롤러를 찾도록 요청하는데, 그 결과로 실행 체인(HandlerExecutionChain)을 리턴해 준다. 이 실행 체인은 1개 이상의 인터셉터가 등록되어 있다면 순차적으로 인터셉터들을 거쳐 컨트롤러가 실행되도록 하고, 인터셉터가 없다면 바로 컨트롤러를 실행한다. 인터셉터는 스프링 컨테이너 내에서 동작하므로 필터를 거쳐 프론트 컨트롤러인 DispatcherServlet이 요청을 받은 이후에 동작하게 되는데, 이러한 호출 순서를 그림으로 표현하면 다음과 같다.



[인터셉터(Interceptor)의 메서드]
인터셉터를 추가하려면 org.springframework.web.servlet의 HandlerInterceptor 인터페이스를 구현(implements)해야 하며, 이는 다음의 3가지 메서드를 가지고 있다.

```
public interface HandlerInterceptor {
    default boolean preHandle(HttpServletRequest request,
                        HttpServletResponse response, Object handler) throws Exception {
        return true;
    }
    default void postHandle(HttpServletRequest request, HttpServletResponse response,
            Object handler, @Nullable ModelAndView modelAndView) throws Exception {
    }
    default void afterCompletion(HttpServletRequest request, HttpServletResponse
                response, Object handler, @Nullable Exception ex) throws Exception {
    }
}
```

- preHandle() 메서드
preHandle 메서드는 컨트롤러가 호출되기 전에 실행된다. 그렇기 때문에 컨트롤러 이전에 처리해야 하는 전처리 작업이나 요청 정보를 가공하거나 추가하는 경우에 사용할 수 있다.

preHandle의 3번째 파라미터인 handler 파라미터는 핸들러 매핑이 찾아준 컨트롤러 빈에 매핑되는 HandlerMethod라는 새로운 타입의 객체로써, @RequestMapping이 붙은 메서드의 정보를 추상화한 객체이다. 또한 preHandle의 반환 타입은 boolean인데 반환 값이 true이면 다음 단계로 진행이 되지만, false라면 작업을 중단하여 이후의 작업(다음 인터셉터 또는 컨트롤러)은 진행되지 않는다.

- postHandle() 메서드
postHandle 메서드는 컨트롤러를 호출된 후에 실행된다. 그렇기 때문에 컨트롤러 이후에 처리해야 하는 후처리 작업이 있을 때 사용할 수 있다. 이 메서드에는 컨트롤러가 반환하는 ModelAndView 타입의 정보가 제공되는데, 최근에는 Json 형태로 데이터를 제공하는 RestAPI 기반의 컨트롤러(@RestController)를 만들면서 자주 사용되지는 않는다. 또한 컨트롤러 하위 계층에서 작업을 진행하다가 중간에 예외가 발생하면 postHandle은 호출되지 않는다.

- afterCompletion() 메서드
afterCompletion 메서드는 이름 그대로 모든 뷰에서 최종 결과를 생성하는 일을 포함해 모든 작업이 완료된 후에 실행된다. 요청 처리 중에 사용한 리소스를 반환할 때 사용하기에 적합하다. postHandler과 달리 컨트롤러 하위 계층에서 작업을 진행하다가 중간에 예외가 발생하더라도 afterCompletion은 반드시 호출된다.

[인터셉터의 등록]

```
package com.example.springedu.config;

import com.example.springedu.interceptor.TestInterceptor;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new TestInterceptor())
                .addPathPatterns("/hello");

        /*
        registry.addInterceptor(인터셉터객체)
                .addPathPatterns("/*") -  모든 Path 적용
                .addPathPatterns("/sample") - /sample Path 에 대해서만 적용
                .excludePathPatterns("/sample"); - /sample Path 에 대해서만 제외
        */
    }
}
```

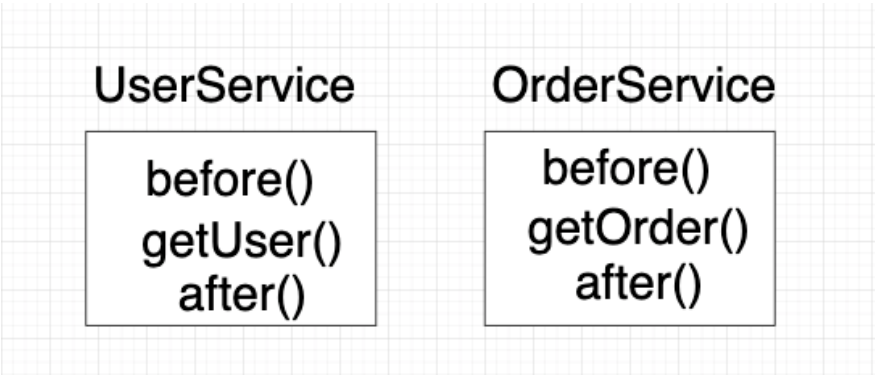
[필터(Filter) vs 인터셉터(Interceptor) 차이 정리 및 요약]

대상	필터(Filter)	인터셉터(Interceptor)
관리되는 컨테이너	서블릿 컨테이너	스프링 컨테이너
스프링의 예외처리 여부	X	O
Request/Response 객체 조작 가능 여부	O	X
용도	• 공통된 보안 및 인증/인가 관련 작업 • 모든 요청에 대한 로깅 또는 감사 • 이미지/데이터 압축 및 문자열 인코딩 • Spring과 분리되어야 하는 기능	• 세부적인 보안 및 인증/인가 공통 작업 • API 호출에 대한 로깅 또는 감사 • Controller로 넘겨주는 정보(데이터)의 가공

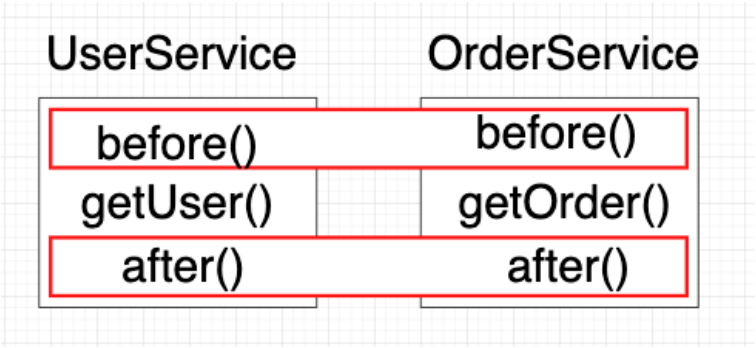
AOP(Aspect-Oriented Programming)

Spring 의 핵심 개념 중 하나인 DI 가 애플리케이션 모듈들 간의 결합도를 낮춘다면, AOP(Aspect-Oriented Programming)는 핵심 로직과 부가 기능(공통 로직)을 분리하여 애플리케이션 전체에 걸쳐 사용되는 부가 기능을 모듈화 하여 재사용할 수 있도록 지원하는 것이다.

Aspect-Oriented Programming 이란 단어를 번역하면 관점(관심) 지향 프로그래밍이 된다. 프로젝트 구조를 바라보는 관점을 바꿔보자는 의미이다.



각 Service 의 핵심기능에서 바라보았을 때 User 과 Order 는 공통된 요소가 없다. 하지만 부가기능 관점에서 바라보면 이야기가 달라진다.



부가기능 관점에서 바라보면 각 Service 의 getXxx() 메서드를 호출하는 전후에 before()와 after() 라는 메서드가 공통으로 사용되는 것을 확인할 수 있다.
기존에 OOP 에서 바라보던 관점을 다르게 하여 부가 기능적인 측면에서 보았을 때 공통된 요소를 분리하자는 것이다. 이때 가로(횡단) 영역의 공통된 부분을 잘라냈다고 하여, AOP 를 크로스 컷팅(Cross-Cutting) 이라고 부르기도 한다.

OOP : 비즈니스 로직의 모듈화
모듈화의 핵심 단위는 비즈니스 로직
AOP : 인프라 혹은 부가기능의 모듈화
대표적인 예 : 모니터링 및 로깅, 동기화, 오류 검사 및 처리, 성능 최적화(캐싱) 등
각각의 모듈들의 주 목적 외에 필요한 부가적인 기능들

간단하게 한 줄로 AOP 를 정리하면, AOP 는 공통된 기능을 재사용하는 기법 이다.
OOP 에선 공통된 기능을 재사용하는 방법으로 상속이나 위임을 사용한다. 하지만 전체 애플리케이션에서 여기저기 사용되는 부가기능들은 상속이나 위임으로 처리하기에는 깔끔한 모듈화가 어렵다. 그래서 등장한 것이 AOP 이다.

AOP 의 장점은 다음과 같다.

- 애플리케이션 전체에 흩어진 공통 기능이 하나의 장소에서 관리되어 유지보수가 좋다.
- 핵심 로직과 부가 기능의 명확한 분리로, 핵심 로직은 자신의 기능에만 집중한다.

[AOP 적용 방식]

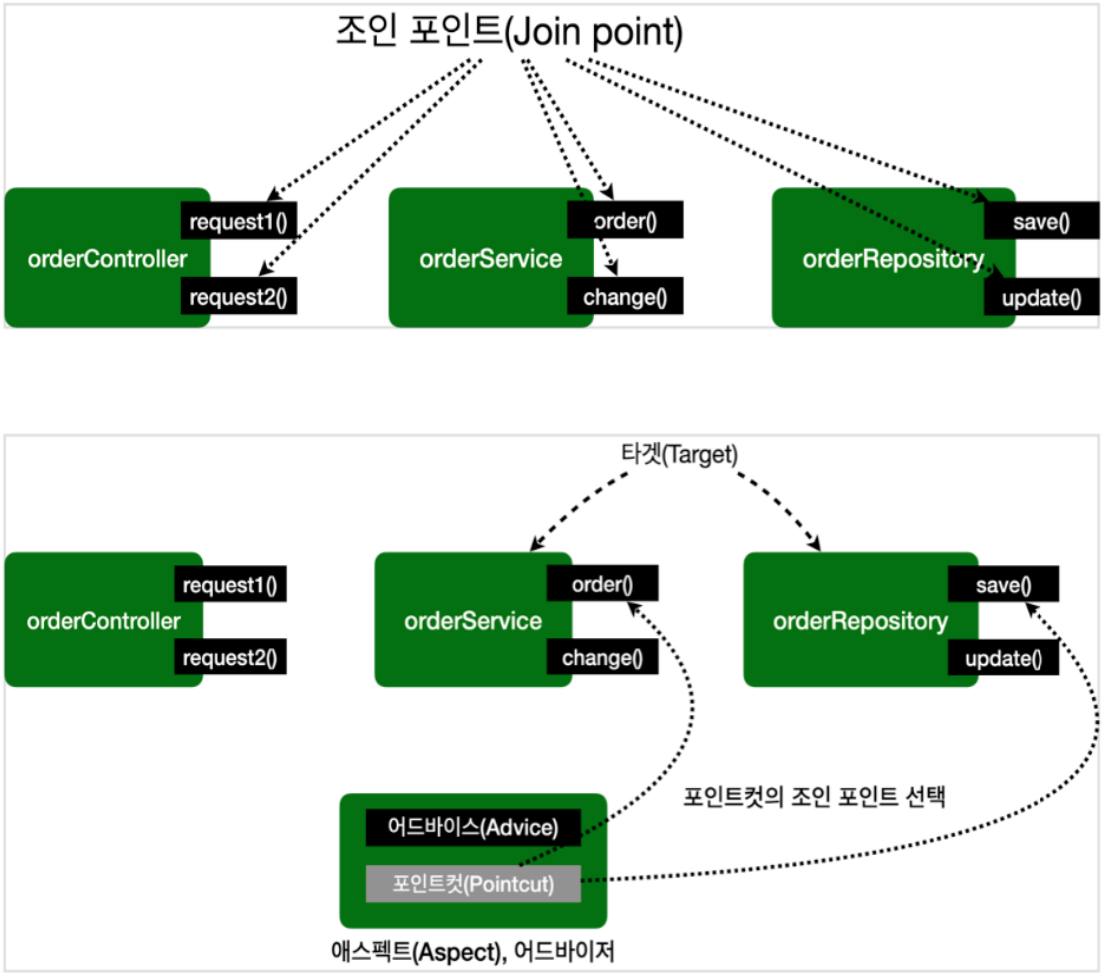
AOP 의 적용 방식은 크게 3 가지 방법이 있다.

- 컴파일 시점
  - .java 파일을 컴파일러를 통해 .class 를 만드는 시점에 부가 기능 로직을 추가하는 방식
  - 모든 지점에 적용하는 것이 가능함
  - AspectJ 가 제공하는 특별한 컴파일러를 사용해야 하기 때문에 특별할 컴파일러가 필요한 점과 복잡하다는 단점
- 클래스 로딩 시점
  - .class 파일을 JVM 내부의 클래스 로더에 올리기 전에 조작하여 부가 기능 로직을 추가하는 방식
  - 모든 지점에 적용하는 것이 가능함
  - 특별한 옵션과 클래스 로더 조작기를 지정해야 하므로 운영하기 어려움
- 런타임 시점
  - 스프링이 사용하는 방식
  - 컴파일이 끝나고 클래스 로더에 이미 다 올라간 후 자바가 실행된 다음에 동작하는 런타임 방식
  - 실제 대상 코드는 그대로 유지되고 프록시를 통해 부가 기능이 적용
  - 프록시는 메서드 오버라이딩 개념으로 동작하기 때문에 메서드에만 적용 가능
    -> 스프링 빈에만 AOP 를 적용 가능
  - 특별한 컴파일러나, 복잡한 옵션, 클래스 로더 조작기를 사용하지 않아도 스프링만 있으면 AOP 를 적용할 수 있기 때문에 스프링 AOP 는 런타임 방식을 사용

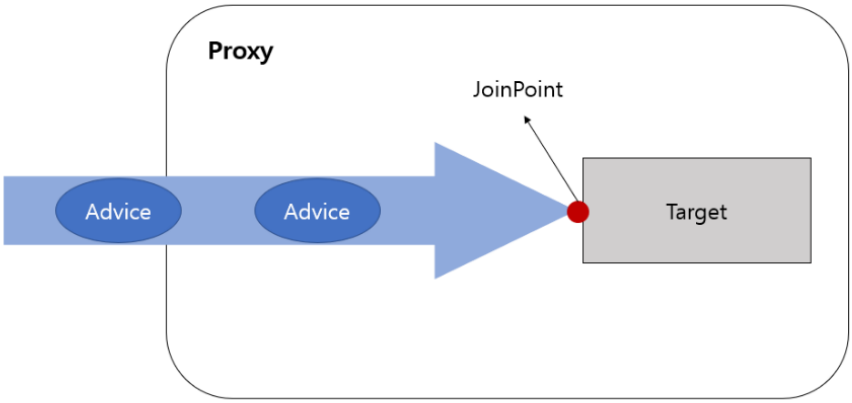
스프링 AOP 는 AspectJ 문법을 차용하고 프록시 방식의 AOP 를 제공한다.

스프링에서는 AspectJ 가 제공하는 애노테이션이나 관련 인터페이스만 사용하고 실제로 AspectJ 가 제공하는 컴파일, 로드타임 위버 등은 사용하지 않는다.

[AOP 용어]



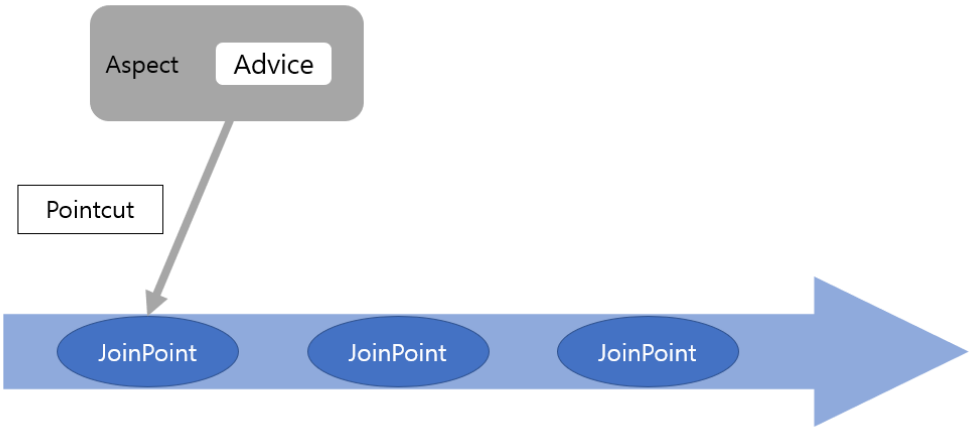
JoinPoint	추상적인 개념으로 advice 가 적용될 수 있는 모든 위치를 말한다. ex) 메서드 실행 시점, 생성자 호출 시점, 필드 값 접근 시점 등등.. 스프링 AOP 는 프록시 방식을 사용하므로 조인 포인트는 항상 메서드 실행 시점
Pointcut	조인 포인트 중에서 advice 가 적용될 위치를 선별하는 기능 스프링 AOP 는 프록시 기반이기 때문에 조인 포인트가 메서드 실행 시점 뿐이 없고 포인트컷도 메서드 실행 시점만 가능
Target	advice 의 대상이 되는 객체 Pointcut 으로 결정됨
Advice	실질적인 부가 기능 로직을 정의하는 곳 특정 조인 포인트에서 Aspect 에 의해 취해지는 조치
Aspect	Advice + Pointcut 을 모듈화 한 것 @Aspect 와 같은 의미
Advisor	스프링 AOP 에서만 사용되는 용어로 Advice + Pointcut 한 쌍
Weaving	Pointcut 으로 결장한 타겟의 JoinPoint 에 Advice 를 적용하는 것
AOP 프록시	AOP 기능을 구현하기 위해 만든 프록시 객체 스프링에서 AOP 프록시는 JDK 동적 프록시 또는 CGLIB 프록시 스프링 부트 2.0 부터 스프링 AOP 의 기본값은 CGLIB 프록시



개발자 입장에서 AOP 를 적용한다는 것은 기존의 코드를 수정하지 않고도 원하는 관심사들을 엮을 수 있다는 점이다. 위 그림에서 Target 에 해당하는 것이 바로 개발자가 작성한 핵심 비즈니스 로직을 가지는 객체이다.

Target 은 순수한 비즈니스 로직을 의미하고, 어떠한 관심사들과도 관계를 맺고 있지 않는다. Target 을 전체적으로 감싸고 있는 존재를 Proxy 라고 한다. Proxy 는 내부적으로 Target 을 호출하지만, 중간에 필요한 관심사들을 거쳐서 Target 을 호출하도록 자동 혹은 수동으로 작성된다. Proxy 의 존재는 직접 코드를 통해서 구현하는 경우도 있지만, 대부분은 스프링 AOP 기능을 이용해서 자동으로 생성되는(auto-proxy) 방식으로 이용한다. JoinPoint 는 Target 객체가 가진 메서드이다. 외부에서의 호출은 Proxy 객체를 통해서 Target 객체의 JoinPoint 를 호출하는 방식이라고 이해할 수 있다.

그림을 통해 Advice 와 JoinPoint 의 관계를 좀 더 상세히 표현하면 다음과 같다.



JoinPoint 는 Target 이 가진 여러 메서드이다. Target 에는 여러 메서드가 존재하기 때문에 어떤 메서드에 관심사를 결합할 것인지를 결정해야 하는데 이 결정을 'Pointcut'이라고 한다.

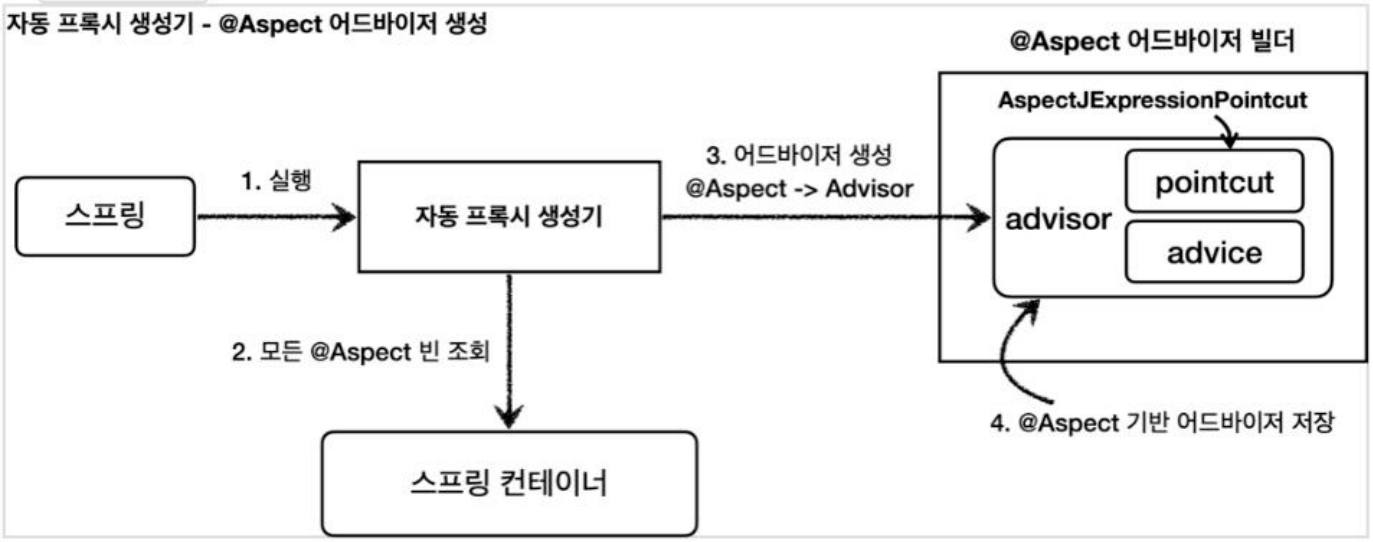
Pointcut 은 관심사와 비즈니스 로직이 결합되는 지점을 결정하는 것이다. 앞의 Proxy 는 이 결합이 완성된 상태이므로 메서드를 호출하게 되면 자동으로 관심사가 결합된 상태로 동작하게 된다.
관심사는 위의 그림에서 Aspect 와 Advice 라는 용어로 표현된다. Aspect 는 관심사 자체를 의미하는 추상명사라고 볼 수 있고, Advice 는 Aspect 를 구현한 코드이다.
과거 스프링에서는 별도의 인터페이스로 구현되고, 이를 클래스로 구현하는 방식으로 제작했지만 스프링 3 버전 이후에는 어노테이션만으로도 모든 설정이 가능하다. Target 에 어떤 Advice 를 적용할 것인지는 XML 을 이용한 설정을 이용할 수 있고, 어노테이션을 이용하는 방식을 이용할 수 있다.

Pointcut 은 Advice 를 어떤 JoinPoint 에 결합할 것인지를 결정하는 설정이다. AOP 에서 Target 은 결과적으로 Pointcut 에 의해서 자신에게는 없는 기능들을 가지게 된다. Pointcut 은 다양한 현태로 사용할 수 있는데 주로 사용되는 설정은 다음과 같다.

execution(@execution) : 메서드를 기준으로 Pointcut 을 설정
within(@within) : 특정한 타입(클래스)을 기준으로 Pointcut 을 설정
this : 주어진 인터페이스를 구현한 객체를 대상으로 Pointcut 을 설정
args(@args): 특정한 파라미터를 가지는 대상들만을 Pointcut 으로 설정
@annotation : 특정한 어노테이션이 적용된 대상들만을 Pointcut 으로 설정

[@Aspect]

@Aspect 애노테이션 을 사용한다면 Advisor 를 더욱 쉽게 구현할 수 있다. 스프링 AOP 를 사용하기 위해서는 다음과 같은 의존성을 추가해야 한다.
implementation 'org.springframework.boot:spring-boot-starter-aop' 의 의존성을 추가하게 되면 자동 프록시 생성기(AnnotationAwareAspectJAutoProxyCreator)를 사용할 수 있게 된다.



자동 프록시 생성기에 의해 @Asepct 에서 Advisor 로 변환된 Advisor 는 @Aspect Advisor 빌더 내부에 저장된다. @Aspect 는 Advisor 를 쉽게 만들 수 있도록 도와주는 역할을 하는 것이지 컴포넌트 스캔이 되는 것은 아닙니다. 따라서 반드시 스프링 빈으로 등록해주어야 한다.

[Advice]

Advice 는 실질적으로 프록시에서 수행하게 되는 로직을 정의하게 되는 곳이다.
스프링에서는 Advice 에 관련된 5 가지 애노테이션을 제공하며 애노테이션은 메서드에 붙이게 된다.
애노테이션의 종류에 따라 포인트컷에 지정된 대상 메서드에서 Advice 가 실행되는 시점 을 정할 수 있다. 또한 속성값으로 Pointcut 을 지정 할 수도 있다.

@Around
- 다른 4 가지 애노테이션을 모두 포함하는 애노테이션
- 메서드 호출 전후 작업 명시 가능
- 조인 포인트 실행 여부 선택 가능
- 입력 값 및 반환 값 변경이나 예외 처리 조작 가능
- 조인 포인트를 여러 번 실행 가능(재시도)

@Before
- 조인 포인트 실행 이전에 실행(실제 target 메서드 수행 전에 실행)
- 입력 값 자체는 변경할 수 없지만 입력 값의 내부에 setter 같은 수정자가 있다면 내부 값은 수정 가능

@AfterReturning
- 조인 포인트가 정상 완료 후 실행(실제 target 메서드 수행 완료 후 실행)
- 반환 값 자체는 변경할 수 없지만 반환 값의 내부에 setter 같은 수정자가 있다면 내부 값은 수정 가능

@AfterThrowing
- 메서드가 예외를 던지는 경우 실행(실제 target 메서드가 예외를 던지는 경우 실행)
- 예외 조작 불가능

@After
- 조인 포인트의 정상, 예외 동작과 무관하게 실행
- (실제 target 메서드가 정상적 수행을 하든 예외를 던지든 수행 이후에 무조건 실행)

JoinPoint 인터페이스가 제공하는 주요 메서드의 기능은 다음과 같다.
- getArgs() : 메서드 인수 반환
- getThis() : 프록시 객체 반환

getTarget() : 대상 객체 반환
getSignature() : 조인되는 메서드에 대한 설명 반환
toString() : 조인되는 방법에 대한 유용한 설명 인쇄

[애노테이션 정리]

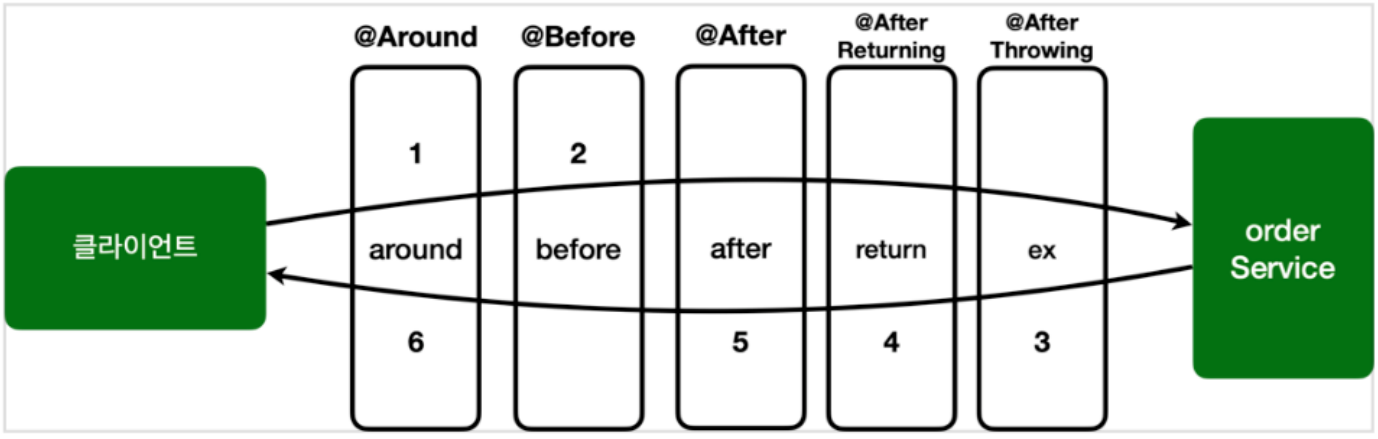
- Aspect 어노테이션을 이용하여 Aspect 클래스에 직접 Advice 와 Pointcut 등을 설정한다
- <aop:aspectj-autoproxy/>를 추가
- Aspect Class 를 <bean>으로 등록

@Aspect : Aspect 클래스 선언
@Before("pointcut")
@AfterReturning(pointcut="", returning="")
@AfterThrowing(pointcut="", throwing="")
@After("pointcut")
@Around("point") : 전반적

Around 를 제외한 나머지 메서드들은 첫번째 인자값으로 JoinPoint 를 가질수 있다
Around 메서드는 인자로 ProceedingJoinPoint 를 가질수 있다

[애노테이션 동작 순서]

동일한 @Aspect 안에서는 위와 같은 우선순위로 동작한다.
즉, 동일한 @Aspect 안에서 여러 개의 Advice 가 존재하는데 타겟 메서드가 여러 Advice 의 대상이 될 경우 다음과 같이 동작한다.
Around -> Before -> AfterThrowing -> AfterReturning -> After -> Around



[Spring AOP 의 @Aspect 예제(1)]

```
@Component
@Aspect
public class MyAspect{
	@Before("execution(public int aop04.*.*())")
	public void b(JoinPoint joinPoint) {
		System.out.println("Before");
	}

	@After("execution(public int aop04.*.*())")
	public void a(JoinPoint joinPoint) {
		System.out.println("After");
	}

	@Around("execution(public int aop04.*.*())")
	public Object around(ProceedingJoinPoint jp) throws Throwable{
		System.out.println("Before Around");
		Object obj = null;
		try {
			obj = jp.proceed();
			System.out.println("Around 리턴 값 : "+obj);
		} catch(Exception e) {
			System.out.println("Around 예외 발생 : "+e.getMessage());
		}
		if (obj == null)
			obj = new Integer(0);
		System.out.println("After Around");
		return obj;
	}
	@AfterThrowing(pointcut="execution(public int aop04.*.*())", throwing="e" )
	public void at(Throwable e){
		System.out.println("AfterThrowing : " + e.getMessage());
	}

	@AfterReturning(pointcut="execution(public int aop04.*.*())", returning="ret" )
	public void ar(Object ret){
		System.out.println("AfterReturning : " + ret);
	}
```

```
        @Before("execution(* *.work2()))")
        public void b1(JoinPoint joinPoint) {
                System.out.println("Before-b1");
        }
        @Before("execution(* *.work3()))")
        public void b2(JoinPoint joinPoint) {
                System.out.println("Before-b2");
        }
}
```

[Advice 순서 지정하기]
애노테이션의 동작 순서는 정의되어 있더라도, 같은 애노테이션에 대한 동작 순서는 보장되지 않다.

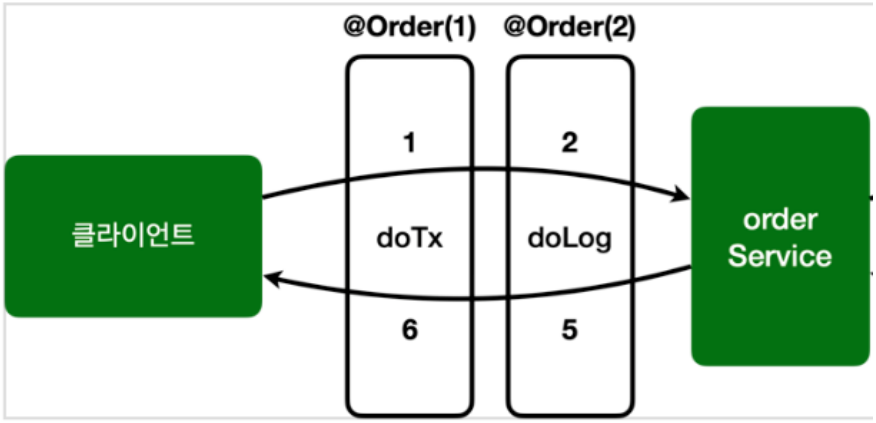
```
@Slf4j
@Aspect
public class AspectV1 {
    @Around("execution(* com.example.mvc.order..*(..))")
    public Object doLog(ProceedingJoinPoint joinPoint)throws Throwable{
        // 생략
    }

    @Around("execution(* com.example.mvc.service..*(..))")
    public Object anotherLog(ProceedingJoinPoint joinPoint)throws Throwable{
        // 생략
    }
}
```

순서를 보장하고 싶다면 @Aspect 적용 단위로 @Order 애노테이션 을 지정해야 한다. 즉, Advice 단위가 아니라 @Aspect 클래스 단위로만 지정이 가능한다. 따라서 하나의 Aspect 안에 여러 Advice 가 존재한다면 처리 순서를 보장할 수 없어 별도의 @Aspect 이 정의된 클래스로 분리해야 한다.

```
@Slf4j
public class AspectV5Order {

    @Aspect
    @Order(1)
    public static class TxAspect {
        @Around("hello.aop.order.aop.Pointcuts.orderAndService()")
        public Object doTx(ProceedingJoinPoint joinPoint) throws Throwable {
```

```
            // 생략
        }
    }

    @Aspect
    @Order(2)
    public static class LogAspect {
        @Around("hello.aop.order.aop.Pointcuts.allOrder()")
        public Object doLog(ProceedingJoinPoint joinPoint) throws Throwable {
            // 생략
        }
    }
}
```

위에서는 내부 static 클래스로 분리했지만, 따로 클래스를 분리해도 된다. 결과적으로 아래 그림처럼 동작하게 된다.



[포인트컷 정의]

Pointcut 은 Advice 가 적용될 위치를 선별하는 기능이다.
스프링 AOP 는 프록시 기반이기 때문에 메서드만 적용 가능하므로 어느 메서드에 적용할 것인지 명시하는 것이다.

- Pointcut 종류

execution - 실질적으로는 가장 많이 사용

execution([접근제어자] 반환타입 [선언타입]메서드이름(파리미터) [예외])

* 패턴을 통해 모든 타입 허용을 표현할 수 있음
.. 을 통해 모든 타입 허용과 파라미터 수가 상관없다는 것을 표현
기본적으로 상위 타입을 명시하면 하위 타입에도 적용 가능
반환 타입이나 파라미터 타입의 경우 정확해야만 매칭됨

within

within 은 클래스 타입을 지정하는 것으로 그 안에 모든 메서드가 매칭됨
execution 에서 타입부분만 사용하는 것과 동일함
정확하게 타입이 맞아야만 동작

bean

스프링 빈의 이름으로 AOP 적용 여부를 지정
스프링에서만 사용할 수 있는 특별한 지시자

[Spring AOP 의 @Aspect 예제(2)]

```
@Aspect
@Component
public class AdviceEx {
        @Before("within(aop09.CoreEx)")
            public void before(){
                    System.out.println("Joinpoint 앞에서 실행되는 Advice");
            }

            @AfterThrowing(pointcut="within(aop09.CoreEx)", throwing="e" )
            public void afterThrowing(Throwable e){
                    System.out.println("예외가 실행될때 호출되는 Advice");
                    System.out.println(e.getMessage());
            }
```

```
	@AfterReturning(pointcut="within(aop09.CoreEx)", returning="ret" )
	public void afterReturing(Object ret){
		System.out.println("Joinpoint 가 정상 종료후 실행되는 Advice");
	}

	@After("within(aop09.CoreEx)")
	public void after(){
		System.out.println("Joinpoint 뒤에서 실행되는 Advice");
	}

	@Around("within(aop09.CoreEx)")
	public Object around(ProceedingJoinPoint  jp) throws Throwable{
		System.out.println("Joinpoint 전에 실행되는 Advice");
		Object result = jp.proceed();
		System.out.println("Joinpoint 후에 실행되는 Advice");
		return result;
	}
}
```

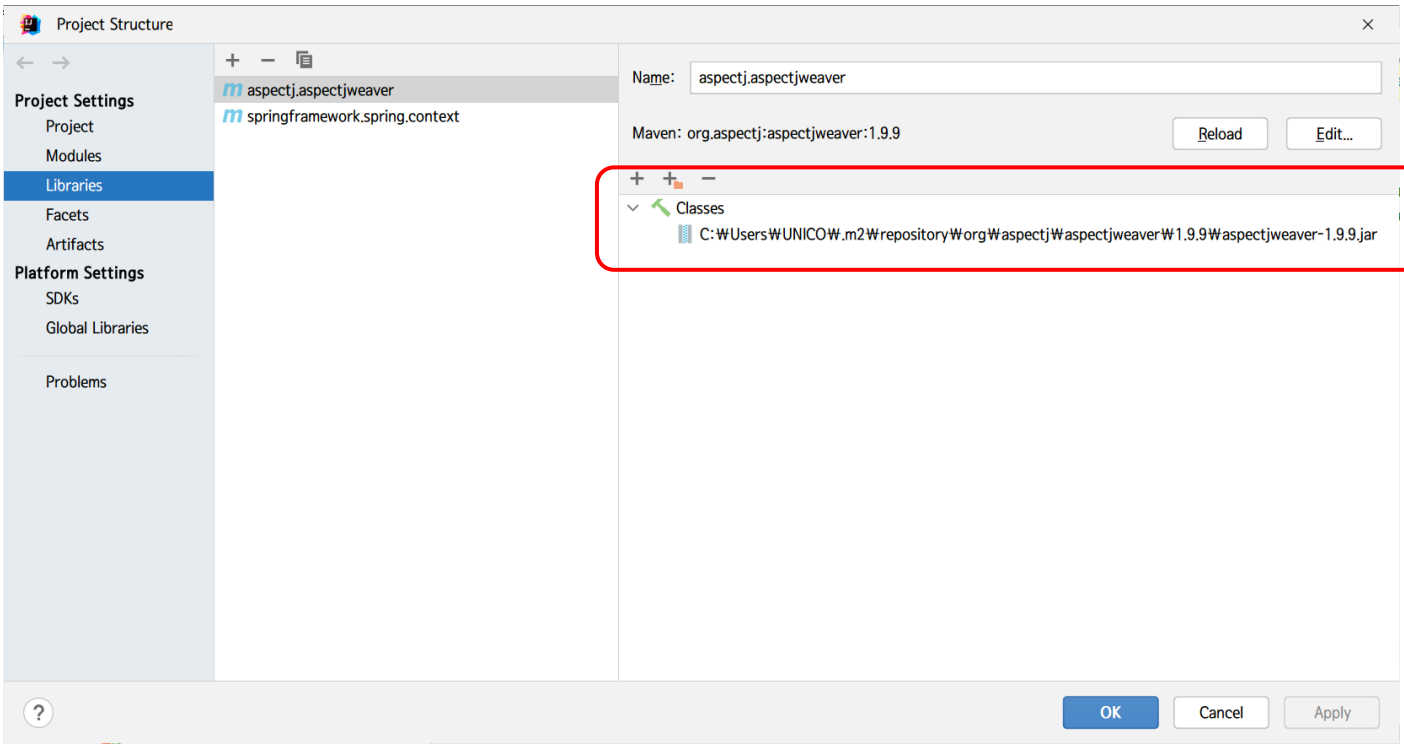
[Spring AOP 의 @Aspect 예제(3)]

```
@Component(value="advice")
@Aspect
public class CalcAdvice {
	@Before("bean(calc)")
	public void before(){
		System.out.println("연산을 시작합니다.");
	}

	@After("bean(calc)")
	public void after(){
		System.out.println("연산을 종료합니다.");
	}

	@Around("bean(calc)")
	public void around(ProceedingJoinPoint pjp) throws Throwable
	{
		System.out.println("** 연산시작 **");
		try {
```

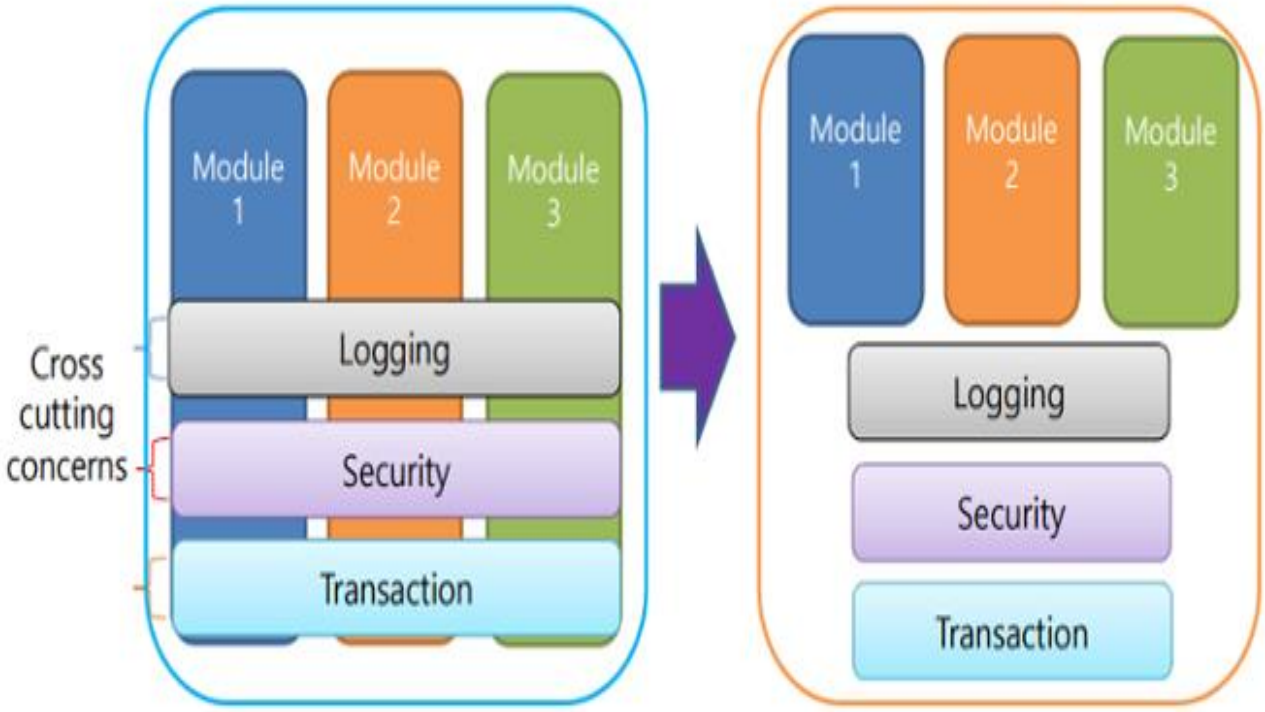
```
                pjp.proceed();
            }catch(Exception e) {
                System.out.println(e.getMessage());
            }
            System.out.println("** 연산종료 **₩n₩n");
        }
}
```

[빈 설정 파일(스프링 디스크립터 파일)]

```
<?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:aop="http://www.springframework.org/schema/aop"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
<context:component-scan base-package="aop12"/>
<aop:aspectj-autoproxy/>
</beans>
```



[Sping AOP 정리]



[AOP 주요용어]]

용어	의미
Advice	공통기능을 담은 구현체
JointPoint	클래스 초기화, 객체 인스턴스화, 메서드 호출, 필드 접근, 예외 발생과 같은 애플리케이션 실행 흐름에서의 특정 포인트를 의미
PointCut	JointPoint의 부분집합으로, Advice가 적용되는 JointPoint Advice를 적용하는 지점(메서드) 스프링은 프록시를 이용해서 AOP를 구현하기 때문에 메서드 호출에 대한 JointPoint만 지원
Weaving	Advice(공통로직)를 핵심로직 코드에 적용하는 것
Aspect	Aspect = Advice(공통기능)+PointCut(메소드 선정 알고리즘) (Advisor 라고도 함)

[Advice의 종류]

종류	설명
Before	대상 객체의 메서드 호출 전 공통기능 실행
After Returning	대상 객체의 메서드가 Exception없이 정상적으로 실행된 이후 공통기능 실행
After Throwing	대상 객체의 메서드가 Exception이 발생했을 때 공통기능 실행
After	Exception여부에 상관없이 항상실행 (finally와 비슷)
Around	메서드 실행 전,후 또는 Exception발생시점에 공통기능 실행 다양한 시점에 원하는 기능을 삽입할 수 있기 때문에 널리 사용된다.