AOP

AOP

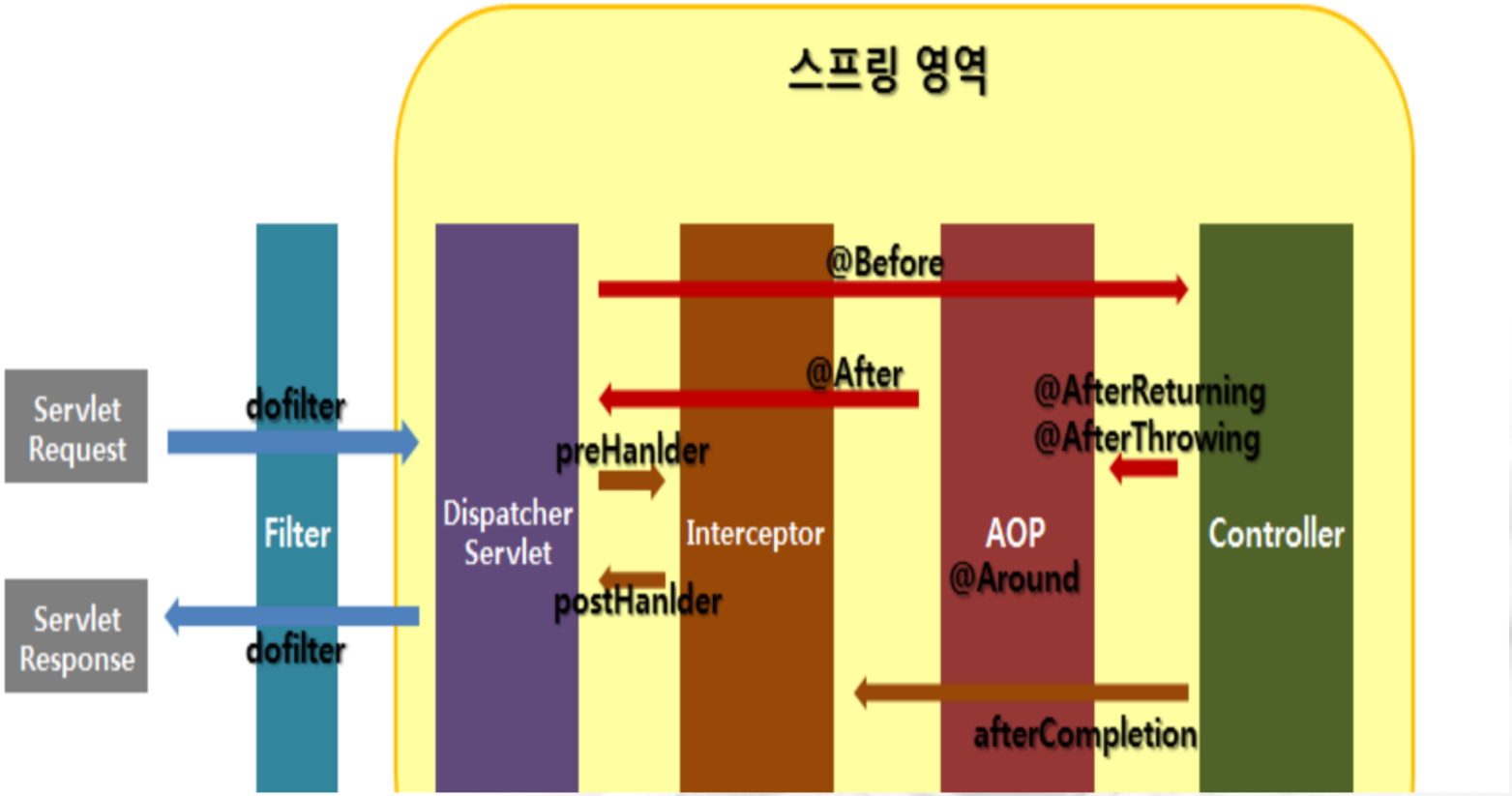
- AOP(Aspect Oriented Programming - 관점 지향 프로그래밍)
  - ✓ 객체 지향 프로그래밍을 보완하는 개념으로 메서드나 클래스를 관점에 따라 분리시켜서 구현하는 프로그래밍 방법
  - ✓ 로그를 출력하는 부분이나 예외를 처리하는 부분처럼 애플리케이션 내에서 공통으로 자주 사용하는 코드(공통 관심사항-cross cutting concern)와 비즈니스 로직(핵심 관심사항-core concern)을 구현하는 부분이 하나의 메서드에 존재할 때 이를 나누어서 구현하는 것
  - ✓ 로그를 출력하는 부분과 비즈니스 로직에 관련된 부분은 아무런 연관성도 없는데 하나의 메서드에 존재한다면 프로그램을 이해하기 어려워질 수 있음
  - ✓ 공통 코드를 분리해서 별도의 클래스에 작성하고 핵심 로직의 앞이나 뒤에서 공통 코드를 불러들여서 사용하는 방식이 AOP
  - ✓ 이를 위해서 별도의 메서드를 직접 호출하는 것이 아니고 설정 파일에 기록해두고 이를 적용해서 클래스를 로딩하거나 로딩한 클래스의 객체를 생성할 때 코드를 삽입하여 사용
  - ✓ AOP를 적용하게 되면 공통 기능 관련 코드가 없기 때문에 적용해야 할 공통 기능이 변경되더라도 핵심 로직은 변경할 필요가 없어짐

AOP

- AOP(Aspect Oriented Programming - 관점 지향 프로그래밍)
  - ✓ 구현 방법
    - Filter
      - Spring Framework 와 무관한 Java EE 의 Spec
      - 인코딩, XSS(Cross-site Scripting) 방어 등에 주로 이용
    - Spring - Interceptor
      - URI 요청 및 응답 시점을 가로채서 전/후 처리를 수행
    - Spring - AOP
      - 비즈니스 로직에서 실행
      - Logging, transaction 처리 등 중복 코드가 발생할 경우 중복을 줄이기 위해 사용되며 메서드 처리 전후 지점에 자유롭게 설정 가능

AOP

- AOP(Aspect Oriented Programming - 관점 지향 프로그래밍)
  - ✓ 구현 방법



Interceptor

- HandlerInterceptor
  - ✓ Controller에서 요청을 처리하기 전이나 요청을 처리한 후 에 동작할 코드를 별도의 코드로 작성해두고 Web Application이 실행 될 때 스프링이 내부적으로 코드를 삽입해서 작업을 할 수 있도록 할 수 있는데 이러한 용도로 사용하는 인터페이스가 HandlerInterceptor
  - ✓ URL을 요청할 때 제어할 목적으로 Java EE에서는 Filter 인터페이스를 제공
  - ✓ Spring을 사용하는 프로젝트에서는 Filter 인터페이스를 잘 사용하지 않는데 이유는 Filter는 Spring Context 외부에서 동작하므로 Spring Context 내의 데이터에 접근이 어렵지만 HandlerInterceptor는 이 작업이 용이하기 때문

Interceptor

- HandlerInterceptor
  - ✓ 인터페이스에는 3가지 메서드가 존재
    - boolean preHandle(HttpServletRequest, HttpServletResponse, Object): 컨트롤러의 요청 처리 메서드를 실행 하기 전에 호출되는데 이 메서드에서 true를 리턴하면 컨트롤러의 요청 처리 메서드를 실행하고 false를 리턴하면 컨트롤러의 요청 처리 메서드를 실행하지 않음
    - void postHandle(HttpServletRequest, HttpServletResponse, Object, ModelAndView): 컨트롤러가 요청 처리 메서드를 정상적으로 수행 한 후에 추가 기능을 구현할 때 사용하는 메서드로 만일 컨트롤러의 요청 처리 메서드에서 예외가 발생하면 이 메서드는 호출되지 않음
    - void afterCompletion(HttpServletRequest, HttpServletResponse, Object, Exception): 클라이언트에 뷰를 전송한 후에 실행되는 메서드로 컨트롤러의 요청 처리 메서드에서 예외가 발생된다면 4번째 매개변수에 그 내용을 전달

Interceptor

❖ Interceptor 설정

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new 인터셉터())
                .addPathPatterns("/*") // 해당 경로에 접근하기 전에 인터셉터가 가로챈다.
                .excludePathPatterns("/boards"); // 해당 경로는 인터셉터가 가로채지 않는다.
    }
}
```

Interceptor

- ❖ Interceptor 적용
  - ✓ Spring Boot Project 생성
    - ● 의존성: spring-web, lombok

Interceptor

- Interceptor 적용
  - ✓ 핸들러 클래스 생성 - MeasuringInterceptor

```
public class MyInterceptor implements HandlerInterceptor {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Override
    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler
    ) throws Exception {
        logger.info("[MYTEST] preHandle");
        return true;
    }
```

Interceptor

- Interceptor 적용
  - ✓ 핸들러 클래스 생성 - MeasuringInterceptor

```
@Override
public void postHandle(
        HttpServletRequest request,
        HttpServletResponse response,
        Object handler,
        ModelAndView modelAndView
) throws Exception {
    logger.info("[MYTEST] postHandle");
}

@Override
public void afterCompletion(
        HttpServletRequest request,
        HttpServletResponse response,
        Object object,
        Exception ex
) throws Exception {
    logger.info("[MYTEST] afterCompletion");
}
}
```

Interceptor

- Interceptor 적용
  - ✓ 핸들러 설정 - WebMvcConfig

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new MyInterceptor())
                .addPathPatterns("/user"); // 해당 경로에 접근하기 전에 인터셉터가
    가로챈다.
    }
}
```

Interceptor

- Interceptor 적용
  - ✓ Controller - HomeController

```
@Controller
@RequestMapping("/")
public class HomeController {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @GetMapping("/user")
    public String users() {
        logger.info("[MYTEST] user!!");
        return "생략..";
    }

    @GetMapping("/board")
    public String boards() {
        logger.info("[MYTEST] board!!");
        return "생략..";
    }
}
```

Interceptor

- Interceptor 적용
  - ✓ 실행 후 브라우저에서 확인
    - localhost:8080/user
    - localhost:8080/board

AOP

- AOP
  - ✓ AOP의 필요성
    - 일반적인 자바 애플리케이션은 웹 계층, 비즈니스 계층, 데이터 계층 등 여러 계층으로 응용 프로그램을 개발
    - 각 계층의 책임은 다르지만 로깅, 트랜잭션, 보안, 인증, 캐싱 등 공통적인 로직이 필요
    - 이러한 공통적인 로직을 횡단 관심사(cross-cutting concern)이라고 부름
    - 공통 로직을 각 계층에서 개별적으로 구현하면, 코드 유지 관리가 어려움
    - 이 문제를 극복하기 위해 AOP는 횡단 관심사를 구현하는 해결책을 제공
  - ✓ AOP의 장점
    - 각 문제에 대한 논리가 소스코드 전체에 흩어져 있는 대신 한 곳에 존재
    - 비즈니스 모듈에는 주요 관심사에 대한 코드만 포함
    - 비즈니스 코드를 수정하지 않고도 추가 동작을 더할 수 있음
  - ✓ AOP는 횡단 관심사의 분리를 통해 공통 기능을 한 곳에 정의하여 모듈성을 증가시킴
  - ✓ 스프링은 내부에서 트랜잭션 관리, 캐싱, 보안 등의 선언적인 서비스를 구현하기 위해 Spring AOP 프레임워크를 제공
  - ✓ 스프링 프레임워크 대신 AspectJ를 애플리케이션에서 AOP 프레임워크로 사용할 수도 있음
  - ✓ Spring AOP vs AspectJ - https://www.baeldung.com/spring-aop-vs-aspectj

AOP

- 구현 방식
  - ✓ AOP의 구현 방식은 Proxy 패턴을 이용해서 구현하게 되는데 외부에서 특정 객체(Target)을 호출하면 실제 객체를 감싸고 있는 바깥쪽 객체(Proxy)를 호출해서 특정 객체에 호출이 전달되는 방식
  - ✓ Proxy는 AOP가 적용된 상태에서 호출을 받아 생성하고 만들어 질 때 Target과 동일한 타입으로 생성되므로 Target과 동일한 방식으로 호출 가능

AOP

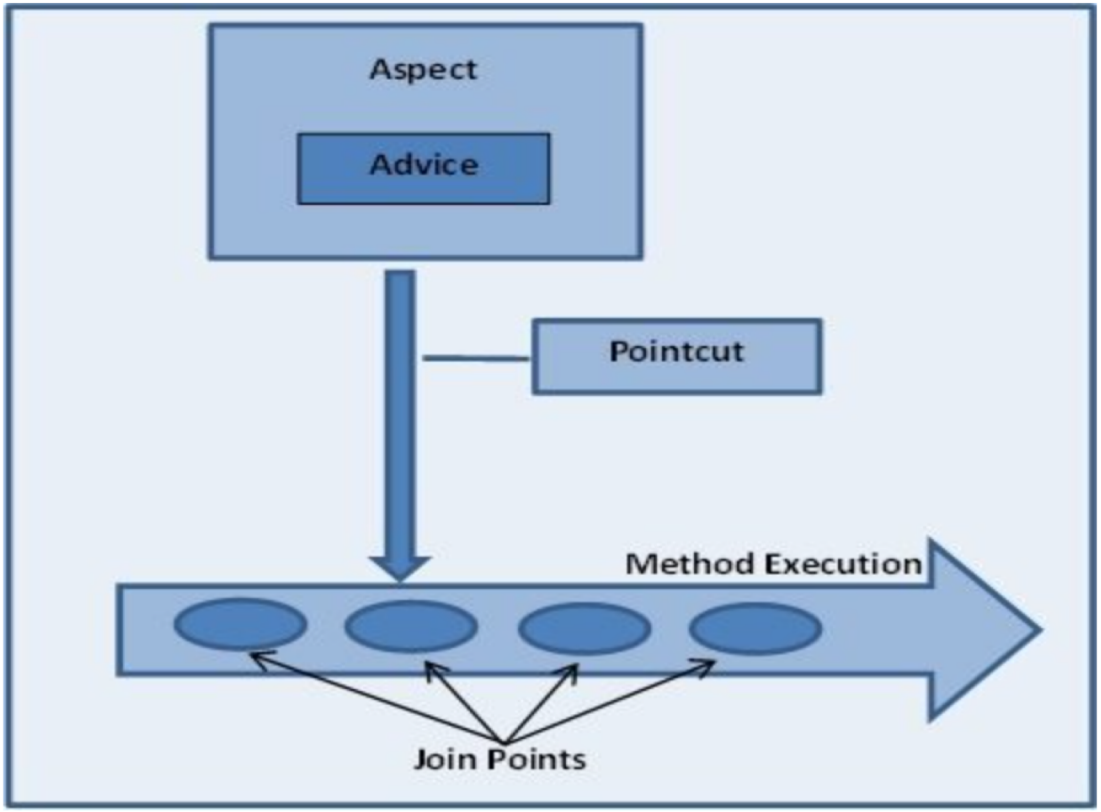
- AOP 용어
  - ✓ Advice
    - 공통 기능의 코드를 의미로 로그 출력, 트랜잭션 관리 등의 코드를 기술
    - 적용할 시점(언제-Around, Before, After Returning, After Throwing, Introduction)을 설정해서 사용
  - ✓ JoinPoints
    - advice가 적용 가능한 지점
    - 조인 포인트는 개발자가 고려해서 만들어 넣을 수 없는 AOP(제품)의 사양
    - 스프링에서는 메서드가 호출될 때 와 메서드가 원래 호출한 곳으로 돌아갈 때 가 어드바이스를 끼워 넣을 수 있는 조인 포인트
  - ✓ Pointcut
    - Advice 와 JoinPoints를 결합하기 위한 설정
    - 정규 표현식 이나 패턴을 이용해서 정의하는데 Spring Framework는 AspectJ pointcut 표현 언어를 사용
  - ✓ Weaving: Advice와 Target을 결합해서 Proxy를 만드는 시점(컴파일 시, 클래스 로딩 시, 런타임 시)
  - ✓ Target: Advice가 적용되는 객체

AOP

- AOP 용어
  - ✓ Aspect
    - 횡단 관심사의 동작과 그 횡단 관심사를 적용하는 소스 코드 상의 포인트를 모은 것
    - 하나 이상의 어드바이스(동작)와 포인트컷(동작을 적용하는 조건)을 조합한 것
    - @Aspect 애너테이션이 달린 일반 클래스를 사용하여 aspect를 구현할 수 있음
  - ✓ Weaving
    - Aspect를 다른 애플리케이션 타입과 연결하는 프로세스
      - 컴파일 시 위빙: 핵심 로직을 구현한 자바 소스 코드를 컴파일 할 때 알맞은 위치에 공통 코드를 삽입하는 방식
      - 클래스 로딩 시 위빙: JVM이 클래스를 로딩할 때 공통 코드를 삽입하는 방식
      - 런타임 시 위빙: 실행 될 때 공통 코드를 호출하는 방식으로 프록시를 이용하여 AOP를 적용하는 것으로 메서드 호출 전후에만 적용 가능

AOP

- AOP



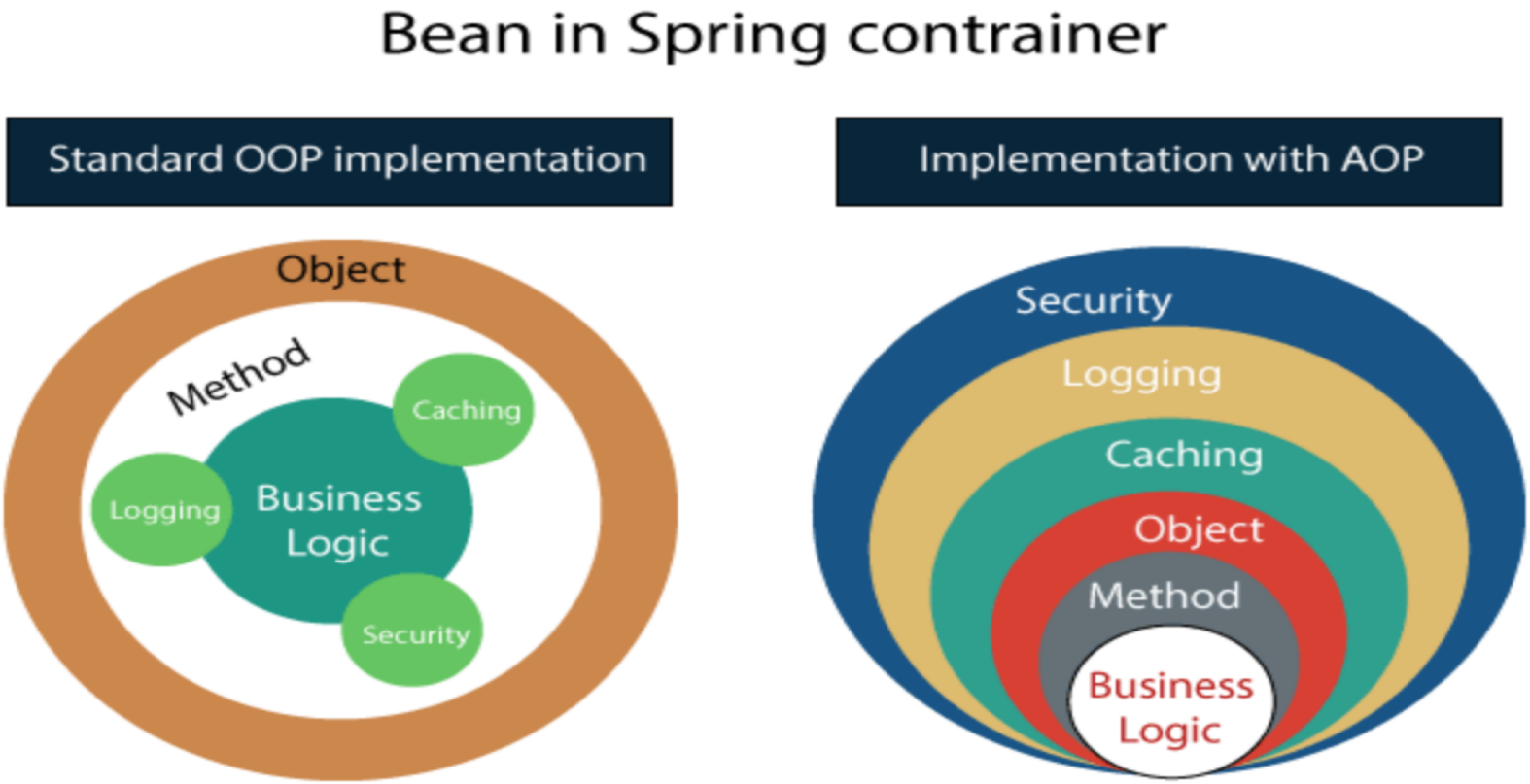
AOP

- AOP

AOP	OOP
Aspect: 포인트컷, 어드바이스 및 속성을 캡슐화하는 코드 단위	Class: 메서드와 속성을 캡슐화하는 코드 단위
Pointcut: 어드바이스가 실행되는 진입점들을 정의	Method Signature: 메서드 바디 실행을 위한 진입점들을 정의
Advice: 횡단 관심사의 구현	Methid bodies: 비즈니스 로직 관심사의 구현
Weaver: 어드바이스로 코드(소스 또는 개체)를 구성	Compier: 소스 코드를 객체 코드르 변경함

AOP

- AOP


Bean in Spring contrainer
Standard OOP implementation
Implementation with AOP
Object
Method
Caching
Logging
Business Logic
Security
Security
Logging
Caching
Object
Method
Business Logic

AOP

❖ AOP

Spring AOP	AspectJ
별도의 컴파일 과정이 필요	AspectJ 컴파일러 필요
메서드 실행 포인트 컷만 지원	모든 포인트 컷을 지원
스프링 컨테이너에서 관리하는 빈에서 구현	모든 도메인 객체에서 구현 가능
메서드 레벨 위빙만 지원	필드, 메서드, 생성자, static initializer, final 클래등을 위빙

AOP

- Spring의 AOP
  - ✓ 스프링에서는 자체적으로 Proxy기반의 AOP를 지원
  - ✓ 스프링의 AOP는 메서드 호출 JoinPoint 만을 지원(필드 값 변경 시 AOP를 사용하고자 할 때는 별도의 AOP 도구를 이용)
  - ✓ AOP의 적용대상이 되는 객체에 직접 접근하는 것이 아니고 Proxy를 통해서 간접적으로 접근하며 Proxy를 이용하지 않고도 AOP 구현 가능
  - ✓ 스프링에서 AOP 구현 방법 – final class에는 적용이 불가
    - XML 기반의 POJO 클래스 또는 MethodInterceptor 인터페이스를 이용한 방법
    - @Aspect Annotation 기반의 구현
  - ✓ 구현 가능한 Advice
    - Before Advice: 대상 객체의 메서드 호출 전에 기능 실행 - @Befor
    - After Returning Advice: 대상 객체의 메서드가 예외 없이 실행한 이후에 기능 실행 - @AfterReturning
    - After Throwing Advice: 대상 객체의 메서드가 예외를 발생한 경우에 기능 실행 - @AfterThrowing
    - After Advice: 대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 기능 실행 - @After
    - Around Advice: 대상 객체의 메서드 실행 전, 후 또는 예외 발생 시점에 기능 실행 - @Around

AOP

- Pointcut 표현식 작성
  - ✓ execution(public void set*(..): 리턴 타입이 void 이고 메서드의 이름이 set으로 시작하고 파라미터가 0개 이상인 모든 public 메서드
  - ✓ execution(* 패키지.*.*(): 패키지 내에서 파라미터가 없는 모든 메서드
  - ✓ execution(* 패키지.*.*(..): 패키지 내에서 파라미터가 0개 이상인 모든 메서드
  - ✓ execution(Integer 패키지.*.*(): 패키지 내에서 리턴 타입이 Integer인 파라미터가 없는 모든 메서드
  - ✓ execution(* get* (*): get으로 시작하고 1개의 파라미터를 갖는 메서드
  - ✓ execution(* get* (*, *): get으로 시작하고 2개의 파라미터를 갖는 메서드
  - ✓ execution(* get* (Integer, *): get으로 시작하고 첫번째 파라미터의 타입이 Integer이고 다른 1개의 파라미터를 갖는 메서드
  - ✓ and 나 or를 이용해서 조합이 가능

AOP

- AOP 적용
  - ✓ build.gradle 파일에 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-aop'
```

AOP

- AOP 적용
  - ✓ SpringBoot Application 클래스에 어노테이션 추가

```
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = true)
public class AopApplication {

    public static void main(String[] args) {
        SpringApplication.run(AopApplication.class, args);
    }

}
```

AOP

- AOP 적용
  - ✓ Domain 클래스를 추가하고 작성

```
@Builder
@Data
public class Employee
{
    private String empId;
    private String firstName;
    private String secondName;
}
```

AOP

- AOP 적용
  - ✓ Service 인터페이스 생성

```
public interface EmployeeService {
    Employee createEmployee(String empId, String fname, String sname);
}
```

AOP

- AOP 적용
  - ✓ ServiceImpl 클래스 생성

```
@Service
public class EmployeeServiceImpl implements EmployeeService {
    public Employee createEmployee( String empId, String fname, String sname) {
        Employee emp = Employee.builder()
                .empId(empId)
                .firstName(fname)
                .secondName(sname)
                .build();
        return emp;
    }
}
```

AOP

- AOP 적용
  - ✓ Controller 클래스 생성

```
@RestController
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;
    @RequestMapping(value = "/add/employee", method = RequestMethod.GET)
    public ResponseEntity<Employee> addEmployee(@RequestParam("empId") String
empId,
                                                @RequestParam("firstName") String firstName,
                                                @RequestParam("secondName") String
secondName) {
        Employee employee = employeeService.createEmployee(empId, firstName,
secondName);

        HttpHeaders headers= new HttpHeaders();
        headers.setContentType(new MediaType("application", "json",
Charset.forName("UTF-8")));

        return new ResponseEntity<>(employee, headers, HttpStatus.OK);
    }
}
```

AOP

- AOP 적용
  - ✓ AOP 적용 클래스 생성

```
@Aspect
@Component
public class EmployeeServiceAspect
{
    @Before(value = "execution(* com.example.aop.service.EmployeeService.*(..)) and
args(empId, fname, sname)")
    public void beforeAdvice(JoinPoint joinPoint, String empId, String fname, String
sname) {
        System.out.println("Before method:" + joinPoint.getSignature());
        System.out.println("Creating Employee with first name - " + fname + ", second
name - " + sname + " and id - " + empId);
    }

    @After(value = "execution(* com.example.aop.service.EmployeeService.*(..)) and
args(empId, fname, sname)")
    public void afterAdvice(JoinPoint joinPoint, String empId, String fname, String
sname) {
        System.out.println("After method:" + joinPoint.getSignature());
        System.out.println("Creating Employee with first name - " + fname + ", second
name - " + sname + " and id - " + empId);
    }
```

AOP

- AOP 적용
  - ✓ AOP 적용 클래스 생성

```
@Pointcut(value= "execution(* com.example.aop.service.EmployeeService.*(..))")
private void logDisplaying() {
}
@Around(value = "logDisplaying()")
public void aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable   {
    System.out.println("The method aroundAdvice() before invokation of the method
" + joinPoint.getSignature().getName() + " method");

    joinPoint.proceed();

    System.out.println("The method aroundAdvice() after invokation of the method "
+ joinPoint.getSignature().getName() + " method");
}
```

AOP

- AOP 적용
  - ✓ AOP 적용 클래스 생성

```
    @AfterReturning(value="execution(* com.example.aop.service.EmployeeService.*(..))
and args(empId, fname, sname)")
    public void afterReturningAdvice(JoinPoint joinPoint, String empId, String fname,
String sname) {
        System.out.println(joinPoint.getSignature().getName() + " 메서드 실행히
정상적으로 완료하였습니다.");
    }
}
```

AOP

- AOP 적용
  - ✓ 실행 한 후 브라우저에서 확인

    http://localhost:8080/add/employee?empId=100&firstName=Ted&secondName=mosby