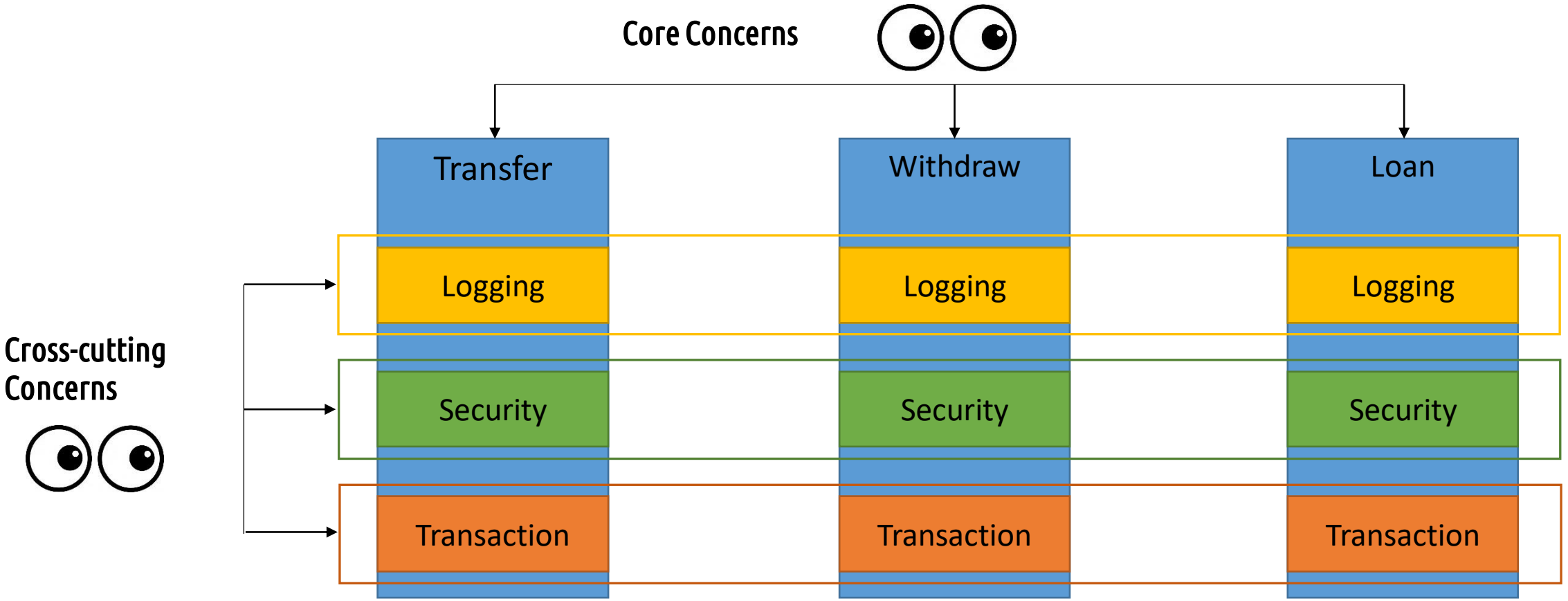
AOP개념 설명

AOP

- **Introduction**



AOP

- **Introduction**

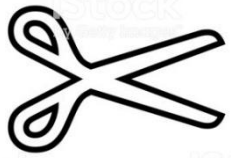
Logging

Security

Transaction

Cross-cutting concern

공통 관심사
부가기능



Separation of Concerns

Transfer

Withdraw

Loan

Core concern

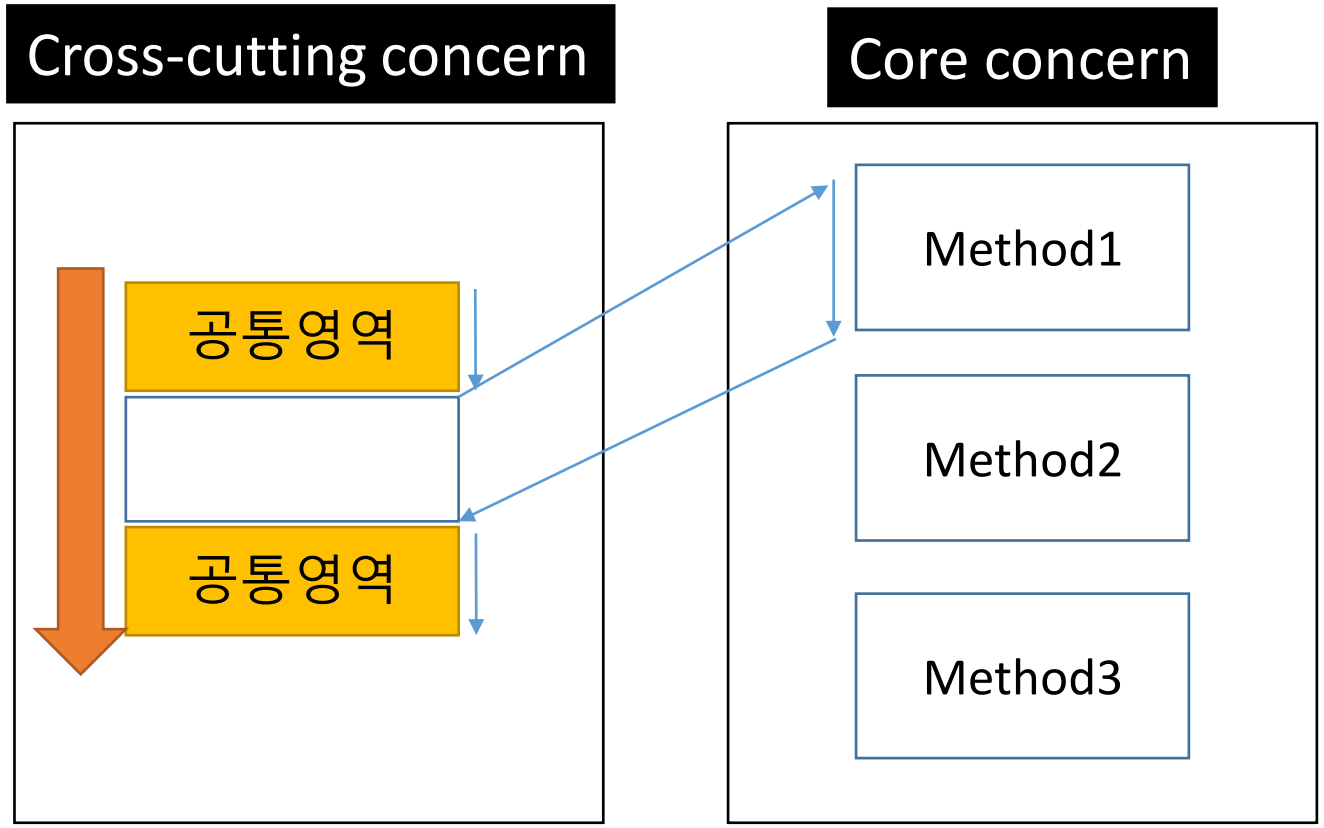
비즈니스 로직
핵심기능

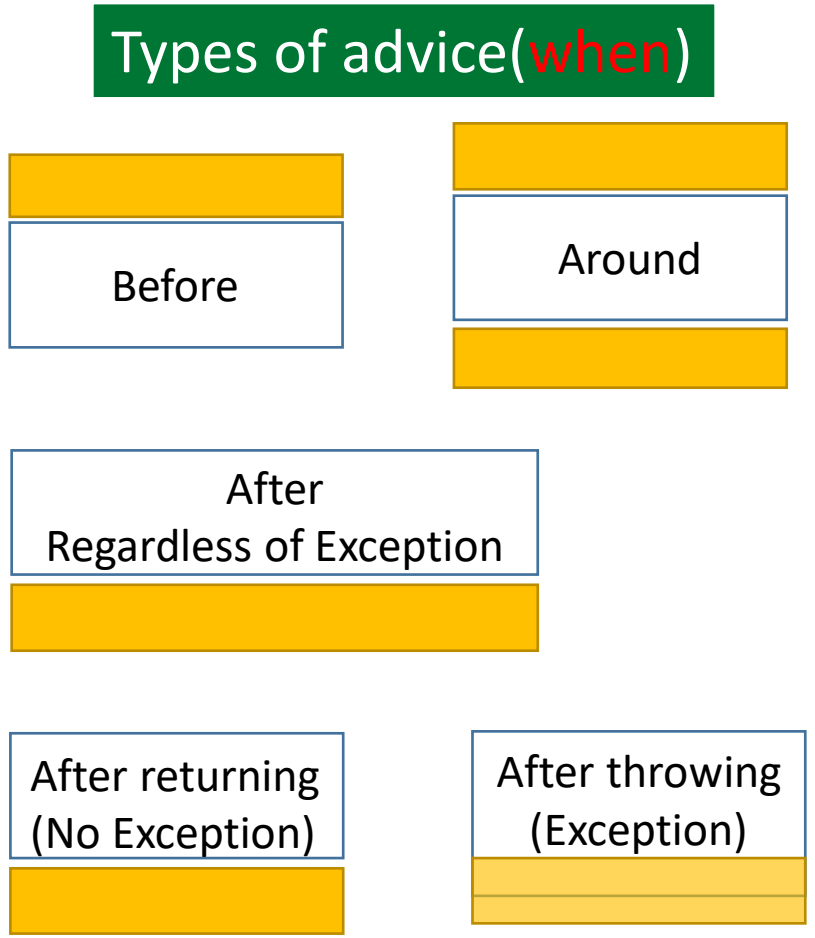
스프링의 @Transactional 어노테이션은 AOP기반으로 만들어짐

AOP

- **Introduction**

Advice: 공통관심이 core concern 어디에 위치하는가

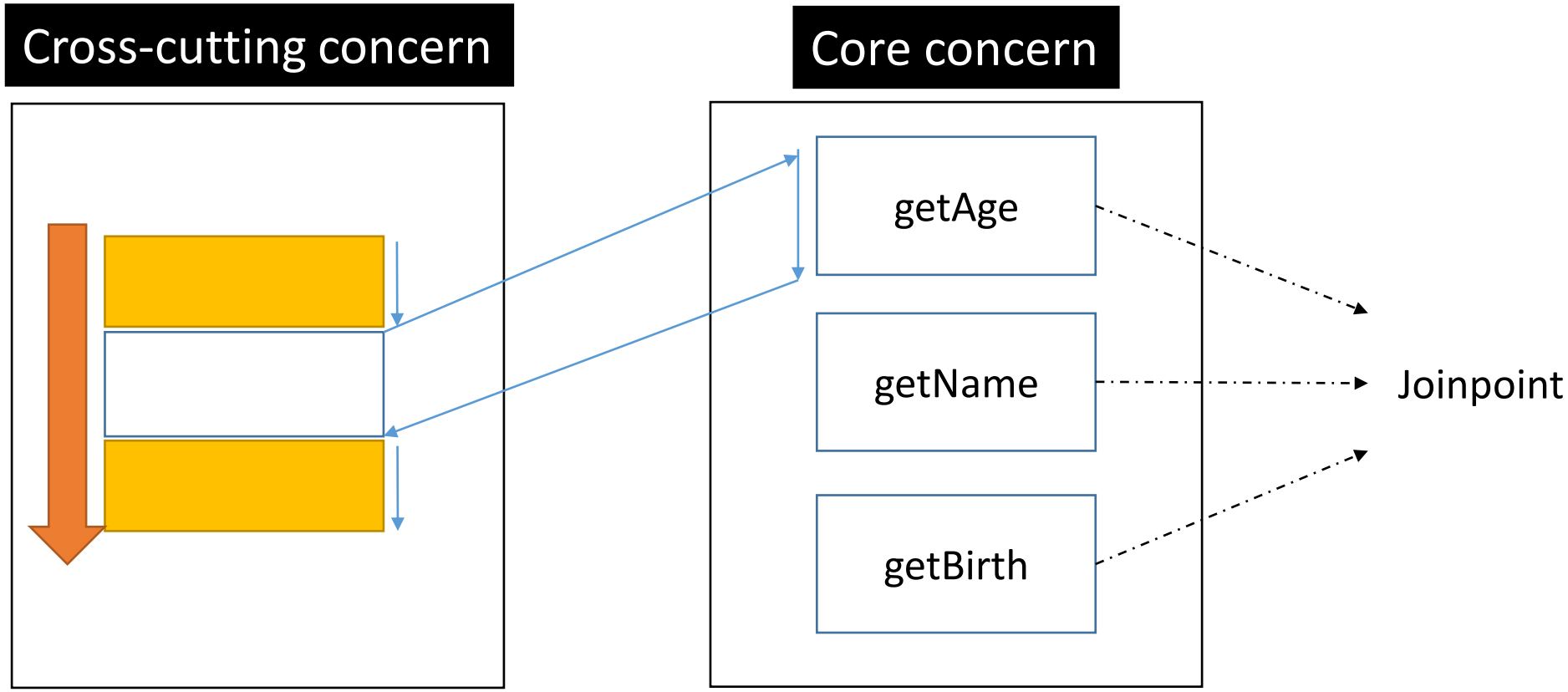




After-returning : (예외 없이) 메서드가 성공 후 cross-cutting concern 실행
After-throwing : 메서드가 예외발생 후 cross-cutting concern 실행

AOP

- **Introduction**



- Joinpoint: target of advice(field, method.. only method is available in spring, field?→AspectJ)

AOP

https://docs.spring.io/spring/docs/2.5.x/reference/aop.html

- **AOP concepts**
  - Aspect
    - a modularization of a concern that cuts across multiple classes
    - aspects are implemented using regular classes or regular classes annotated with the @Aspect annotation
  - Join point: AOP가 적용될 대상(메소드)
    - a point during the execution of a program, such as the execution of a method or the handling of an exception
    - In Spring AOP, a join point always represents a method execution
  - Advice: Join point에서 AOP가 적용되는 시점
    - action taken by an aspect at a particular join point
    - "around," "before" and "after" advice.

AOP

https://docs.spring.io/spring/docs/2.5.x/reference/aop.html

- **AOP concepts**
  - Pointcut: AOP가 적용될 메소드를 표현하는 마치 정규식과 유사한 표현식
  - 정의한 aspect가 모든 메소드에 적용될 수도 있고, 특정 패키지 하위에 적용될 수도 있고..
    - a predicate that matches join points
    - Advice is associated with a pointcut expression and runs at any join point matched by the pointcut
    - Spring uses the AspectJ pointcut expression language by default

```java
@Service
@Slf4j
public class EmployeeService {

    public String getEmployeeNameFromId(String id){
        return "Test Name From Service";
    }
}
```

```java
@RestController
@Slf4j
@RequestMapping("/api")
public class Employee {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping(value = "/get/employee/name/{id}")
    public String getEmployeeName(@PathVariable String id){
        if (StringUtils.isBlank(id)){
            return null;
        }
        return employeeService.getEmployeeNameFromId(id);
    }
}
```

[출처] https://blog.fisenko.net/measure-method-execution-time-using-spring-aop/

```java
@Aspect
@Component
@Slf4j
@ConditionalOnExpression("${aspect.enabled:true}")
public class ExecutionTimeAdvice {
```

Advice **PointCut**

```java
    @Around("@annotation(com.mailshine.springboot.aop.aspectj.advise.TrackExecutionTime)")//이 어노테이션이 붙은 곳에 적용
    public Object executionTime(ProceedingJoinPoint point) throws Throwable {
        long startTime = System.currentTimeMillis();
        Object object = point.proceed();//core concern실행
        long endtime = System.currentTimeMillis();
        log.info("Class Name: "+ point.getSignature().getDeclaringTypeName() +". Method Name: "+ point.getSignature().getName() +
". Time taken for Execution is : " + (endtime-startTime) +"ms");
        return object;
    }
}
```

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TrackExecutionTime {
}
```

```java
@Service
@Slf4j
public class EmployeeService {

    @TrackExecutionTime
    public String getEmployeeNameFromId(String id){
        return "Test Name From Service";
    }
}
```

[출처] https://medium.com/@mailshine/spring-boot-measure-execution-time-using-spring-aop-and-aspectj-4590d7bd6bb2

AOP

- **지시자(PCD, AspectJ pointcut designators)의 종류**
  - execution : 가장 정교하게 결합점(joint point)을 정의
  - within : 타입패턴으로 결합점을 정의
  - bean : bean이름으로 결합점을 정의

- **리턴 타입 지정**
  - * : 모든 리턴 타입 허용
  - void: 리터 타입이 void인 메소드
  - !void: 리턴 타입이 void가 아닌 메소드

AOP

execution(접근제어자_반환형_패지키를 포함한 클래스 경로_메소드_메소드파라미터)

- **패키지 지정**
  - com.kit.dormitory: 정확하게 com.kit.dormitory 패키지만 선택
  - com.kit.dormitory.. : com.kit.dormitory 패키지 및 하위 패키지

- **클래스 지정**
  - UserBO: 정확하게 UserBO 클래스만
  - *BO: 이름이 BO로 끝나는 클래스만

- **메소드 지정**
  - *(. .): 모든 메소드
  - update*(. .) : 메소드명이 update로 시작하는 모든 메소드

AOP

execution(접근제어자_반환형_패지키를 포함한 클래스 경로_메소드_메소드파라미터)

- **매개변수 지정**
  - (..): 모든 매개변수
  - (*): 반드시 1개의 매개변수를 가지는 메소드
  - (com.kit.dormitory.member.Member): Member를 매개변수로 가지는 메소드만(fully qualified name)
  - (Integer,..): 한 개 이상의 매개변수를 갖되, 첫 번째 매개변수의 타입이 Integer인 메소드만
  - (Integer,*): 반드시 두 개의 매개변수를 갖되, 첫 번째 매개변수의 타입이 Integer인 메소드만

AOP

- **PointCut 예시**
  - **execution(접근제어자_반환형_패지키를 포함한 클래스 경로_메소드_메소드파라미터)**

"execution(public void get*())"

public형의, 반환이 없고, 이름이 get으로 시작하고, 파라미터가 없는 모든 메소드

"execution(* * (..))"

첫 번째 * → 접근제어자와 반환형 모두 상관하지 않고 적용
두 번째 * → 어떠한 경로에 존재하는 클래스도 상관하지 않겠다
.. → 파라미터가 몇 개가 존재하던지 상관 없음

"execution(* com.java.ex.Car.accelerate())"

첫 번째 * → 접근제어자와 반환형 모두 상관하지 않고 적용
com.java.ex.Car 클래스의 파라미터가 없는 accelerate 메소드에 적용

"execution(* com.java..*.*)"

첫 번째 * → 접근제어자와 반환형 모두 상관하지 않고 적용
.. → 해당 패지키를 포함한 모든 하위패키지에 적용

AOP

- **PointCut 예시**
  - **execution(접근제어자_반환형_패지키를 포함한 클래스 경로_메소드_메소드파라미터)**

"within(com.java.ex.*)"

com.java.ex. 하위의 모든 클래스의 모든 메소드에 적용

"within(com.java.ex..*)"

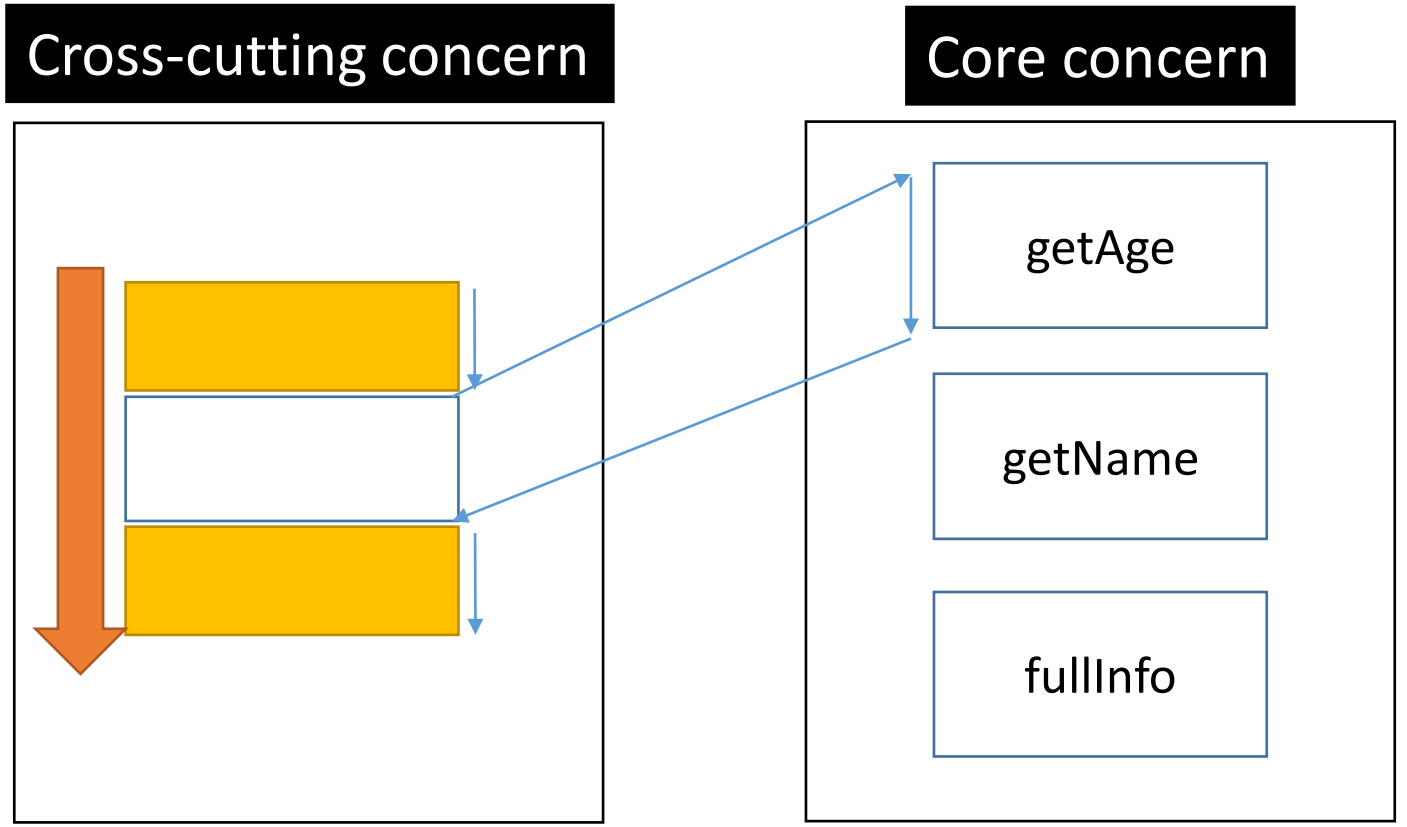
com.java.ex 패키지 및 하위 패키지를 포함한 모든 캘래스의 메소드에 적용

"bean(bean(car))"

car라는 이름의 bean에게 적용

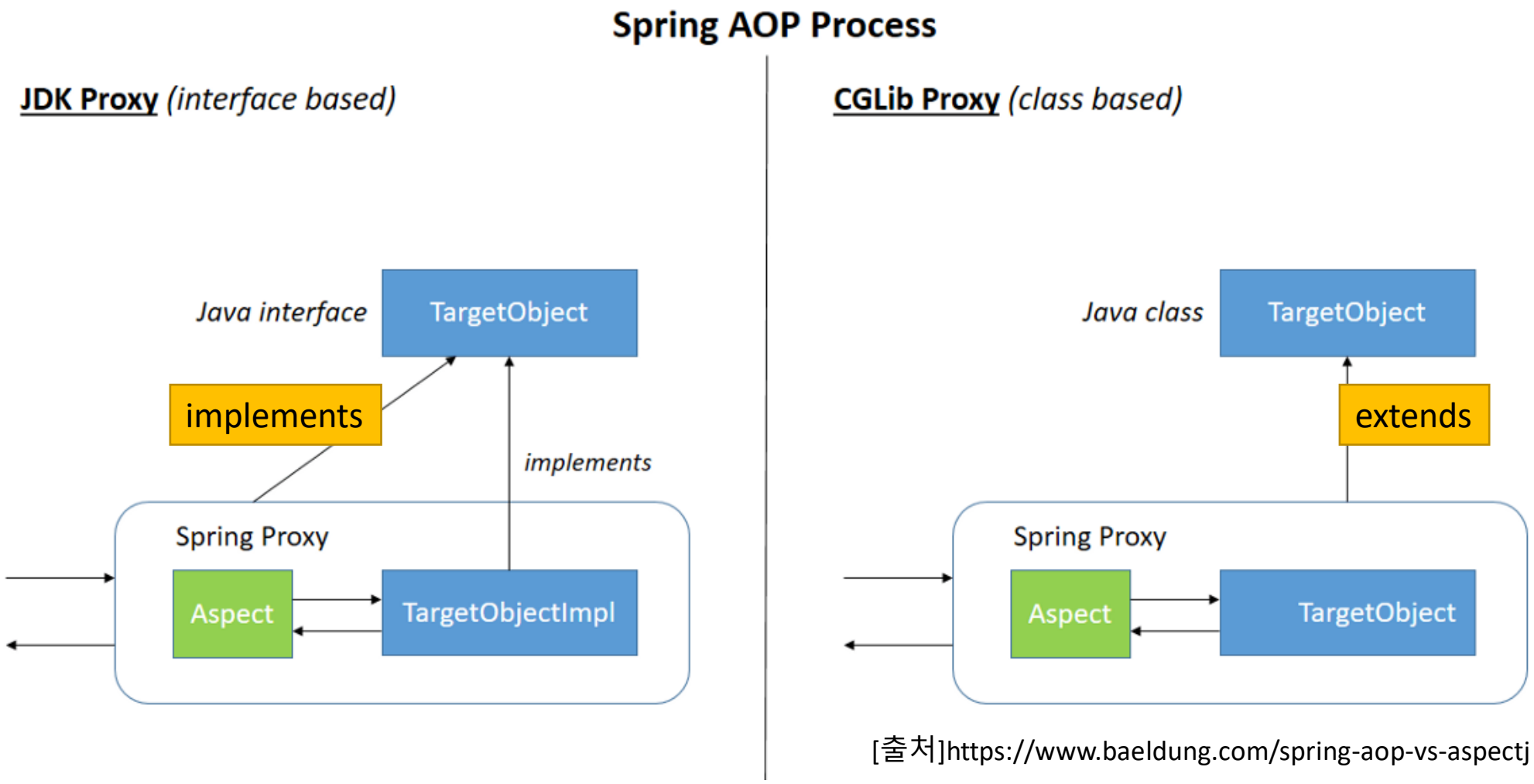
AOP

- **Weaving**
  - Weaving is the process of linking aspects with target
  - When?
    - Compile Time(CTW)
    - Load Time(LTW)
    - Run time(RTW):스프링이 사용하는 방법



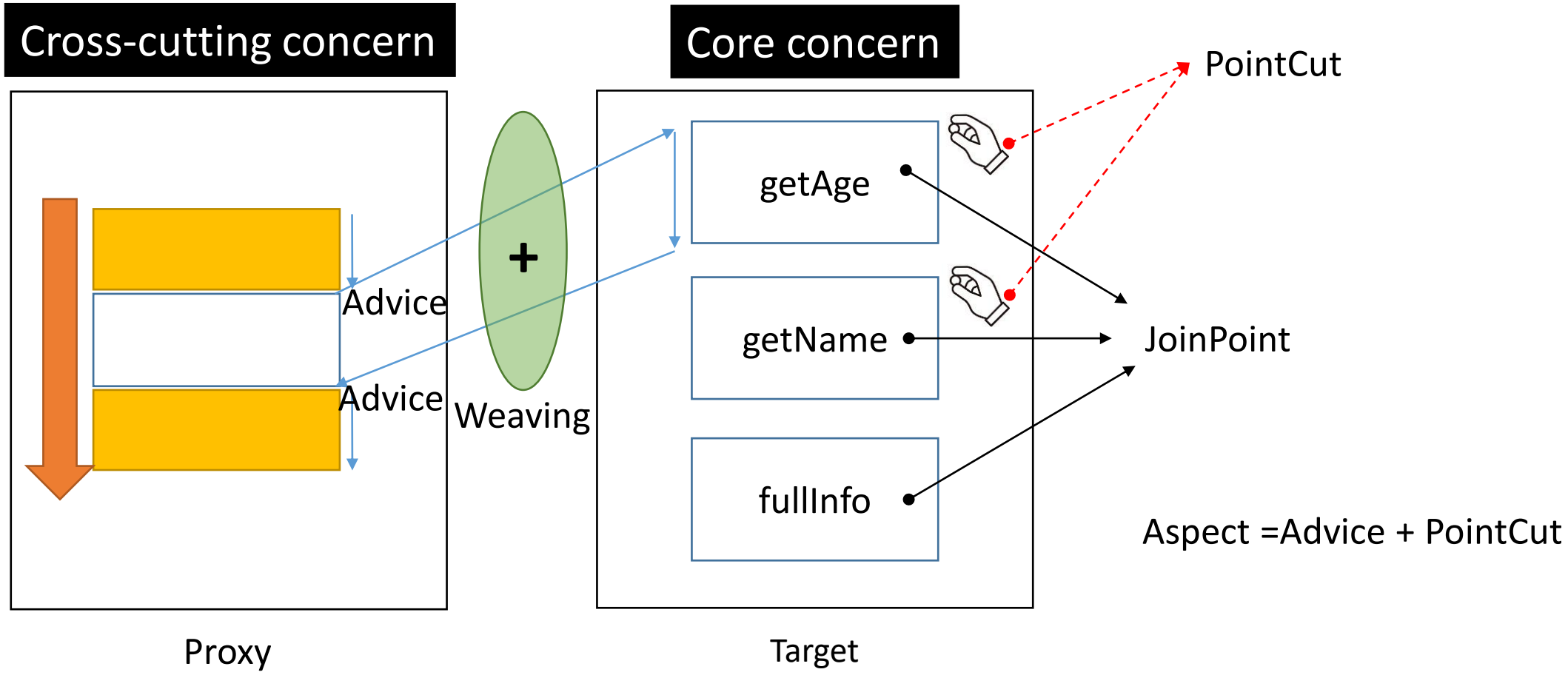
AOP

- **Spring에서 AOP는 두 가지 방법으로 구현**
  - Dynamic Proxy
  - CGLIB



[출처]https://www.baeldung.com/spring-aop-vs-aspectj

AOP(Terms)



JDK Dynamic Proxy(interface)
CGLib Proxy(extends)
`(proxyTargetClass = true)`