

스프링 5

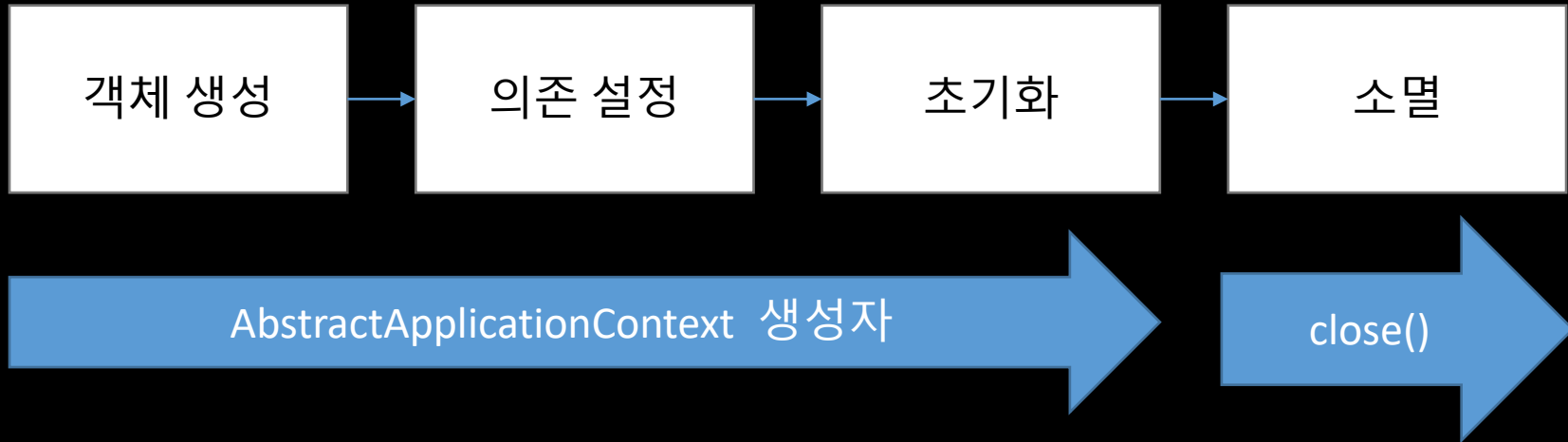
Chapter 6-7

정승혜

CH 6

빈 라이프 사이클과 범위

스프링 컨테이너/객체의 라이프 사이클



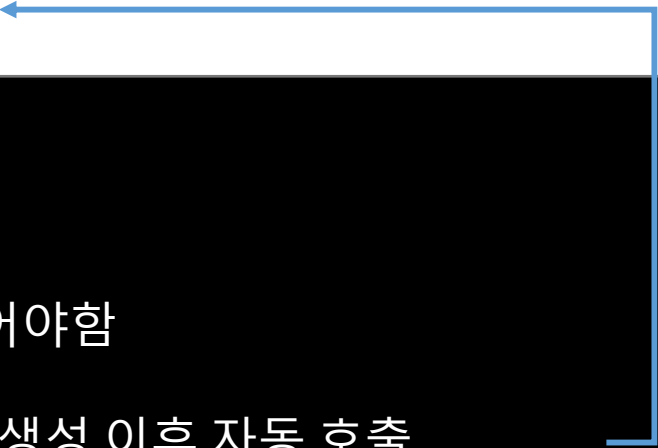
스프링 Bean 객체의 초기화/소멸 : 스프링 인터페이스

`org.springframework.beans.factory.InitializingBean`
⇒ **afterPropertiesSet()**

`org.springframework.beans.factory.DisposableBean`
⇒ **destroy()**

커스텀 메서드

```
@Bean(initMethod = "connect", destroyMethod = "close")  
Public Client client2() {  
....
```



1. 메서드는 파라미터가 없어야함
2. 초기화 메서드는 빈 객체 생성 이후 자동 호출,
빈 객체 생성 코드 내부에 메서드를 호출하여
중복되지 않도록 주의

프로토타입 빈의 생성

```
@Bean  
@Scope("prototype")  
Public Client client2() {  
....
```

getBean() 메서드를 통해 객체를 얻을 때마다
새로운 객체가 생성 됨

** 주의사항 **

프로토타입 범위 빈은 완전 라이프 사이클 따르지 x
스프링 컨테이너는 (생성, 프로퍼티 설정, 초기화)는 수행하지만,
컨테이너를 종료한다고 해서 프로토타입 빈 객체의 소멸까지 책임 x

⇒ **Prototype 범위 빈 사용 시에는 객체 소멸 코드를 직접 해야함**

CH 7

AOP 프로그래밍

Proxy 객체

- 핵심 기능은 다른 객체 (대상 객체)에 위임하고 부가적인 기능을 제공
- 기존 코드를 변경하지 않아도 됨
- 코드의 중복을 줄임

```

public class ExeTimeCalculator implements Calculator{
    private Calculator delegate;

    public ExeTimeCalculator(Calculator delegate) {
        this.delegate = delegate;
    }

    @Override public long factorial(long num) {
        long start = System.nanoTime();
        long result = delegate.factorial(num);
        long end = System.nanoTime();
        System.out.printf("%s.factorial(%d) 실행시간 = %d\n",
            delegate.getClass().getSimpleName(), num, (end-start));
        return result;
    }
}

```

Proxy : 접근 제어 관점, decorator: 기능 추가와 확장에 초점

AOP

- Aspect Oriented Programming (관심 지향)
- Aspectjweaver (AOP 어노테이션) 사용
- 핵심 기능/공통 기능 구현 분리 -> 핵심 기능 코드 수정없이 공통 기능 적용 가능

AOP = 핵심 기능에 공통 기능을 삽입

HOW?

1. 컴파일 시점 코드에
2. 클래스 로딩 시점의 바이트 코드에



스프링 AOP 지원 X
AspectJ와 같은 AOP 전용도구
필요

3. 런타임에 프록시 객체를 생성해서
=> 스프링 AOP 제공, 실제 객체 기능을 실행하기 전,후에 공통 기능 호출

스프링 AOP

- 핵심 기능은 다른 객체 (대상 객체)에 위임하고 부가적인 기능을 제공
- 기존 코드를 변경하지 않아도 됨
- 코드의 중복을 줄임
- 프록시 객체를 자동으로 만들어 줌. 직접 구현 x
- 프록시를 이용해 AOP를 구현하기 때문에 메서드 호출에 대한 Joinpoint만 지원

AOP 주요 용어	
Aspect	여러 객체에 공통으로 적용되는 기능
Advice	언제 공통 관심 기능을 핵심 로직에 적용할 것인지를 정의
JoinPoint	Advice를 적용 가능한 지점
Pointcut	joinpoint의 부분집합, 실제 Advice가 적용되는 Joinpoint를 나타냄
Weaving	Advice를 핵심 로직 코드에 적용하는 것

스프링에서 구현 가능한 Advice 종류

Before Advice	대상 객체 메서드 호출 전
After Returning Advice	대상 객체의 메서드가 익셉션 없이 실행된 이후
After Throwing Advice	대상 객체의 메서드를 실행하는 도중 익셉션이 발생한 경우
After Advice	익셉션 발생여부에 상관없이 대상 객체의 메서드 실행 후
Around Advice	대상 객체의 메서드 실행 전,후 또는 익셉션 발생 시점

1. Aspect 클래스 설정

`@Aspect` => 현재 클래스가 공통으로 적용될 기능이라 명시

```
public class ExeTimeAspect {
```

`@Pointcut("execution(public * chap07..*(..))")` => 공통 기능을 적용할 대상 명시
execution 명시자

```
private void publicTarget() {  
}
```

`@Around("publicTarget()")` => 공통 기능을 구현한 메서드에
언제 공통 기능을 적용할 건지 Advice 명시 : Around
publicTarget() 메서드에 정의한 Pointcut에 공통 기능 적용

```
public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
```

```
...
```

1. Aspect 클래스 설정

```
@Around("publicTarget()")
public Object measure(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.nanoTime();
    try {
        Object result = joinPoint.proceed();
        // 실제 대상 객체의 메서드를 호출(기준)
        return result;
    } finally {
        long finish = System.nanoTime();
        Signature sig = joinPoint.getSignature(); // 호출 메서드 정보
        System.out.printf("%s.%s(%s) 실행 시간 : %d ns\n",
            joinPoint.getTarget().getClass().getSimpleName(), sig.getName(), // 대상 객체를 구함
            Arrays.toString(joinPoint.getArgs()), (finish-start)); // 파라미터 목록
    }
}
```

2. Configuration 클래스 설정

- @Bean을 통해 객체로 등록해주기

PreceedingJoinPoint

- String getSignature() : 호출되는 메서드에 대한 정보
- Object getTarget() : 대상 객체를 구함
- Object[] getArgs() : 파라미터 목록을 구함

org.aspectj.lang.Signature 인터페이스

- String getName() : 호출되는 메서드의 이름을 구함
- String toLongString() : 호출되는 메서드를 완전하게 표현한 문장을 구함
(리턴타입, 파라미터 타입)
- String toshortString() : 호출되는 메서드를 축약해서 표현한 문장을 구함
(default : 메서드 이름만)

프록시 생성 방식

```
public class ExeTimeCalculator implements Calculator{  
    private Calculator delegate;  
    ....
```

⇒ 인터페이스를 상속하게 되면, 프록시는 인터페이스를 이용해 프록시를 생성함
= Calculator에 대해 프록시 생성

⇒ ExeTimeCalculator, 즉 원래 클래스에 대해 프록시를 생성하고 싶을 때

```
@Configuration  
@EnableAspectJAutoProxy(proxyTargetClass = True) // 추가  
public class AppCtx {  
    ...
```


Advice 적용 순서

- 한 Pointcut에 여러 Advice 를 적용할 수 있다
- 예제) CacheAspect 프록시 -> ExeTimeAspect 프록시 -> Calculator(factorial)

⇒ 순서를 바꾸고 싶다면 => @Order() 어노테이션 사용

1

```
Object result = joinPoint.proceed();  
cache.put(num,result);  
System.out.printf("CacheAspect : Cache에 추가[%d]\n",num);  
return result;
```

4

```
@Around("publicTarget()")  
public Object measure(ProceedingJoinPoing joinPoint) throws Throwable {  
    long start = System.nanoTime();  
    try {  
        Object result = joinPoint.proceed();  
        return result;  
    } finally {  
        ... System.out....  
    }  
}
```

2

3

```
Public class RecCalculator implements Calculator {
```

```
    @Override  
    public long factorial(long num){  
        ...  
    }  
}
```

@Around의 Pointcut 설정과 @Pointcut 재사용

- 한 Pointcut이 아니라 Around어노테이션에 execution 명시자를 직접 지정할 수 있음

Ex) @Around("execution(public * chap07 ..*(..))")

- 만약 여러 Aspect에서 공통으로 사용하는 Pointcut이 있다면 별도 클래스 정의시 편리

```
public class CommonPointcut {
```

```
    @Pointcut("execution ...")  
    public void commonTarget() {  
        ...  
    }
```

```
public class CacheAspect{
```

```
    @Around("CommonPointcut.commonTarget()")  
    public Object execute(...  
        ....
```