스프링 기본편 정리

▼ 2. 스프링 핵심 원리 이해1 - 예제 만들기, 3 **스프링 핵심 원리 이해**2 - **객체 지향 원리 적용**

스프링부트 3.2부터 Gradle 옵션

주의! 스프링 부트 3.2 부터 Gradle 옵션을 선택하자.

스프링 부트 3.2 부터 앞서 Build and run using에 앞서 설명한 IntelliJ IDEA를 선택하면 몇가지 오류가 발생한다.

따라서 스프링 부트 3.2를 사용한다면 다음과 같이 IntelliJ IDEA가 아니라 Gradle을 선택해야 한다.

주요 오류 및 원인

1. intellij idea를 사용해 빌드 및 실행할 경우, 기본적으로 -parameters 옵션이 활성화 되지 않아서 메서드 시그니처에서 파라미터 이름을 추출할 때 문제 발생 → gradle을 사용하면 이 옵션이 기본적으로 활성화되어 있음
2. gradle 플러그인 호환성 문제: spring boot 3.x는 자바17이상 요구

동시성 이슈와 ConcurrentHashMap

❓1. hashmap과 concurrenthashmap의 차이는?

> 참고: `HashMap`은 동시성 이슈가 발생할 수 있다. 이런 경우 `ConcurrentHashMap`을 사용하자.

concurrenthashmap은 자바에서 멀티스레드 환경에서 안전하게 사용할 수 있도록 설계된 map 인터페이스 구현체

1. thread-safe: 여러 스레드가 동시에 concurrenthashmap에 접근해도 데이터 일관성 유지(세부화된 락과 CAS(compare-and-swap) 메커니즘을 통해 이루어짐

   CAS(compare-and-swap) 매커니즘은 멀티스레드 환경에서 동기화를 위해 사용되는 원자적 연산

   특정 메모리 위치의 값을 비교하고, 조건이 충족되면 새로운값으로 교체하는 과정을 수행

전통적인 락을 사용하지 않고도 데이터의 일관성을 유지하며, 비차단 동기화를 제공함

- 장점: CAS는 락을 사용하지 않으므로 데드락 위험이없고, 스레드가 실패 시 재시도 할수 있어, 대기 상태 없이 빠르게 처리 가능
- 단점: CAS가 실패했을 경우, 스레드는 반복적으로 재시도해야 하므로 CPU자원을 낭비, 메모리위치의 값이 A→B→A로 변경되면, CAS는 이를 감지하지 못할 가능성이 있음

2. 동시 읽기/쓰기 지원: 읽기 작업은 락 없이 수행되므로 빠름, 쓰기 작업은 특정 데이터 버킷 단위로 락을 걸어 처리하므로, 전체 맵에 락을 거는 hashtable보다 성능이 뛰어남
3. null키와 값 허용 불가: concurrenthashmap은 null키나 값을 허용하지 않음. 동시성 환경에서 데이터 불일치 문제를 방지하기 위한 설계

내부 동작 원리

1. java8이전: concurrenthashmap은 내부적으로 여러 세그먼트로 나뉘어져 있었으며, 각 세그먼트는 독립적인 락을 가짐, 특정 키에 접근할때 해당 키가 속한 세그먼트만 락을 걸어 동시성을 보장
2. java8이후: 세그먼트 대신 버킷 단위로 락을 관리함, 읽기 작업은 락 없이 수행되며, 쓰기 작업은 특정 버킷에만 락을 걸어 처리함

OCP(open closed principle), DIP(dependency inversion principle)원칙

OCP란 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다

DIP란 객체에서 어떤 클래스를 참조해서 사용해야 하는 상황이 생긴다면, 그 클래스를 직접 참조하는 것이 아니라 그 대상의 상위 요소(추상 클래스나 인터페이스)로 참조하라는 원칙이다

확장을 하려면 기존 코드를 변경해야 하고 이 점을 극복하기 위해 다형성을 활용한다

```java
public class MemberService {
    private MemberRepository memberRepository = new MemoryMemberRepository();
```

```java
public class MemberService {
    //private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();
```

MemberService 클라이언트가 구현 클래스를 직접 선택하고 있음

구현 객체를 변경하려면 클라이언트 코드도 변경해야 함

다형성을 활용했지만 OCP를 지킬 수 없었음 → 이를 해결하기 위해 객체를 생성하고, 연관관계를 맺어주는 별도의 조립, 설정자가 필요함

또한, MemberService 클라이언트가 구현 클래스를 직접 선택하며 DIP를 위반하고 있음

다형성 만으로 OCP, DIP를 지킬 수 없고 이는 스프링의 DI컨테이너와 의존관계 주입을 통해 해결한다

```java
 public class OrderServiceImpl implements OrderService {

      private final MemberRepository memberRepository = new MemoryMemberRepository();
      private final DiscountPolicy discountPolicy = new FixDiscountPolicy();

      @Override
      public Order createOrder(Long memberId, String itemName, int itemPrice) {
        Member member = memberRepository.findById(memberId);
    int discountPrice = discountPolicy.discount(member, itemPrice);
        return new Order(memberId, itemName, itemPrice, discountPrice);
      }
}
```

위 코드는 추상(인터페이스) 뿐만 아니라 구체(구현) 클래스도 의존하고 있다

❓2. 이는 현재 OrderServiceInpl는 OrderService와 할인 정책을 결정하는 책임 또한 가지고 있음 →무엇을 위반?

▼ 정답

SRP 위반

❓ 3. 해결 방안은

▼ 정답

DIP를 위반하지 않도록 인터페이스에만 의존하도록 의존관계를 변경하면 된다

```
public class OrderServiceImpl implements OrderService {
      //private final DiscountPolicy discountPolicy = new RateDiscountPolicy();
      private DiscountPolicy discountPolicy;
}
```

그런데 구현체가 없는데 어떻게 코드를 실행할 수 있을까?

누군가가 클라이언트인 OrderServiceImpl에 DiscountPolicy의 구현 객체를 대신 생성하고 주입해주어야 한다

❓ 4. 어떻게?

▼ 정답

애플리케이션의 전체 동작 방식을 구성(config)하기 위해 구현 객체를 생성하고, 연결하는 책임을 가지는 별도의 설정 클래스를 만들자

```
public class AppConfig {
      public MemberService memberService() {
          return new MemberServiceImpl(new MemoryMemberRepository());
}
      public OrderService orderService() {
          return new OrderServiceImpl(
            new MemoryMemberRepository(),
                new FixDiscountPolicy());
 }
```

```java
public class OrderServiceImpl implements OrderService{

    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;

    public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy discountPoli
        this.memberRepository = memberRepository;
        this.discountPolicy = discountPolicy;
    }

    @Override
    public Order createOrder(Long memberId, String itemName, int itemPrice) {
        Member member = memberRepository.findById(memberId);
        int discountPrice = discountPolicy.discount(member, itemPrice);

        return new Order(memberId, itemName, itemPrice, discountPrice);
    }
}
```

AppConfig는 생성한 객체 인스턴스의 참조(레퍼런스)를 생성자를 통해서 주입(연결)해준다

AppConfig의 등장으로 OrderServiceImpl은 구체 클래스를 의존하지 않음

AppConfig의 등장으로 애플리케이션이 크게 사용 영역과 객체를 생성하고 구성하는 영역으로 분리되었다

AppConfig처럼 객체를 생성하고 관리하면서 의존관계를 연결해주는 것을 Ioc컨테이너, DI컨테이너라고 한다

생성자

❓ 5. 생성자에 대해 설명하시오

인스턴스가 생성될때 호출되는 인스턴스 초기화 메소드

인스턴스 변수의 초기화 작업에 주로 사용되며, 인스턴스 생성 시에 실행되어야 할 작업을 위해 사용된다

```
Card c = new Card();
```

연산자 new에 의해서 메모리(heap)에 Card클래스의 인스턴스가 생성된다

생성자 Card()가 호출되어 수행된다

연산자 new의 결과로, 생성된 Card인스턴스의 주소가 반환되어 참조 변수 c에 저장된다

생성자에서 다른 생성자 호출하기

- 생성자의 이름으로 클래스 이름 대신 this를 사용
- 한 생성자에서 다른 생성자를 호출할때는 반드시 첫줄에서만 호출이 가능함

왜 다른 생성자를 호출할 때 첫 줄에서만 호출이 가능하도록 할까?

호출된 다른 생성자 내에서도 멤버 변수들의 값을 초기화를 할 것이므로 다른 생성자를 호출하기 이전의 초기화 작업이 무의미해질 수 있기 때문이다

this: 참조변수로 인스턴스 자기 자신을 가리킴

static 메소드(클래스 메소드)에서는 인스턴스 멤버들을 사용할 수 없는 것처럼, this 역시 사용할 수 없음

static 메소드는 인스턴스를 생성하지 않고도 호출될 수 없는것처럼, this 역시 사용할 수 없음

static 메소드는 인스턴스를 생성하지 않고도 호출될 수 있으므로 static 메소드가 호출된 시점에 인스턴스가 존재하지 않을 수도 있기 때문이다

생성자를 포함한 모든 인스턴스 메소드에는 자신이 관련된 인스턴스를 가리키는 참조 변수 this가 지역변수로 숨겨진 채로 존재한다

스프링 컨테이너

❓6. 스프링 컨테이너의 역할과 정의

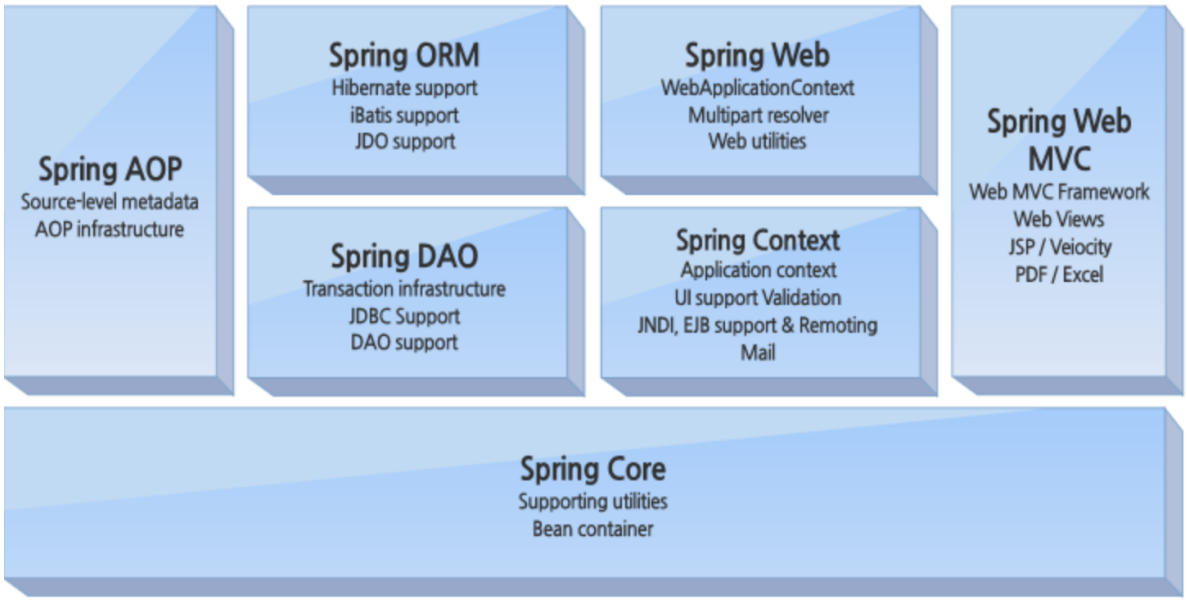
applicationContext를 스프링 컨테이너라고 함

스프링 컨테이너는 @Configuration이 붙은 AppConfig를 설정 정보로 사용함 여기서 @Bean이 붙은 메소드를 모두 호출해서 반환된 객체를 스프링 컨테이너에 등록한다

스프링 컨테이너에 등록된 객체를 스프링 빈이라고 함

▼ 4. 스프링 컨테이너와 스프링 빈, 5. 싱글톤 컨테이너

스프링 컨테이너란 스프링 프레임워크의 핵심이며 스프링 빈의 생명 주기를 관리하고 spring 프레임워크의 특징인 Ioc, DI을 제공해주는 역할을 함



spring core가 spring container을 의미함

말 그대로 핵심이며 bean factory라는 IOC패턴을 적용하여 객체 구성부터 의존성 처리의 역할을 한다

bean factory container vs applicationcontext

❓7. bean factory container vs applicationcontext의 차이

bean factory container

bean factory는 스프링 설정 파일에 등록된 bean객체를 생성하고 관리하는 기본적인 기능만을 제공

container가 구동될 때 bean객체를 생성하는 것이 아니라, 클라이언트의 요청에 의해서 bean객체가 사용되는 시점에 객체를 생성하는 방식을 사용하고 있음

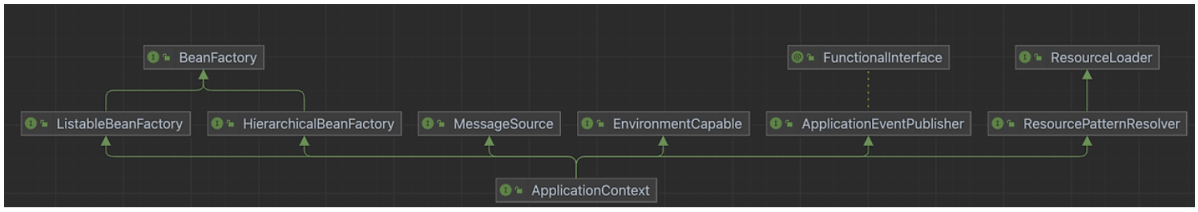
bean factory를 상속받은 applicationContext를 spring container라고 한다

applicationcontext

```java
package org.springframework.context;

import org.springframework.beans.factory.HierarchicalBeanFactory;
import org.springframework.beans.factory.ListableBeanFactory;
import org.springframework.beans.factory.config.AutowireCapableBeanFactory;
import org.springframework.core.env.EnvironmentCapable;
import org.springframework.core.io.support.ResourcePatternResolver;
import org.springframework.lang.Nullable;

public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, Hi
                MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
...
}
```



bean 객체를 생성하고 관리하는 기능, 트랜잭션 관리, 메시지 기반의 다국어 처리, AOP처리 등등 많은 부분 지원

컨테이너가 구동되는 시점에 객체들을 생성하는 pre-loading방식을 따름

xml설정 파일에서는 <component-scan/>, 자바 설정 파일에서는 @ComponentScan을 이용해 스캔 범위를 지정하여 @Component 어노테이션이 붙은 빈으로 등록될 준비를 마친 클래스들을 스캔하여 빈으로 등록해줌

만약 빈으로 등록하기 위한 클래스 정의시 @Configuration이 빠진다면 bean scope가 singleton으로 생성도지 못한다

❓ 8. 스프링이 다양한 컨테이너 설정 형식을 지원할 수 있는 이유는?

▼ 정답

이렇게 스프링이 다양한 컨테이너 설정 형식을 지원할 수 있는 것은 beandefinition이라는 추상화 덕분이다(역할과 구현을 분리)

❓ 9. 스프링 없는 순수한 DI컨테이너인 AppConfig는 요청을 할때마다 객체를 새로 생성해서 메모리 낭비가 심하다 → 해결방안으로 ??? 패턴을 사용한다

▼ 정답

싱글톤

```
public class SingletonService {

    //1. static 영역에 객체를 딱 1개만 생성해둔다.
    private static final SingletonService instance = new SingletonService();

    //2. public으로 열어서 객체 인스턴스가 필요하면 이 static 메서드를 통해서만 조회하도록 허용한다.
    public static SingletonService getInstance() {
        return instance;
    }

    //3. 생성자를 private으로 선언해서 외부에서 new 키워드를 사용한 객체 생성을 못하게 막는다.
    private SingletonService() {
    }

    public void logic() {
        System.out.println("싱글톤 객체 로직 호출");
    }
}
```

static 영역에 객체 instance를 미리 하나 생성해서 올려두고, getInstance() 호출해서 같은 인스턴스를 반환

생성자를 private으로 막아서 외부에서 new키워드로 객체 인스턴스가 생성되는것을 막는다

그런데, 싱글톤 패턴에도 문제가 있음!

❓ 10. 싱글톤 패턴에는 어떤 문제가 있을까?

▼ 정답

- 의존관계상 클라이언트가 구체 클래스에 의존함 → DIP위반
- 클라이언트가 구체 클래스에 의존해서 OCP 위반

- private 생성자로 자식 클래스를 만들기 어려움

❓ 11. 싱글톤 객체를 생성하고 관리하는 기능을 ??? ?????라고 한다

▼ 정답

싱글톤 레지스트리

❓ 12. 싱글톤 방식의 주의점

▼ 정답

- 무상태로 설계해야 함
  - 특정 클라이언트에 의존적인 필드가 있으면 안된다
  - 특정 클라이언트가 값을 변경할 수 있는 필드가 있으면 안된다
  - 가급적 읽기만 가능해야함
- 필드 대신에 자바에서 공유되지 않는, 지역변수, 파라미터, threadlocal을 사용해야 함

1. 지역변수: 메소드 내부에서 선언되어 해당 메소드가 실행되는 동안에만 유효하며, 메소드 호출이 끝나면 메모리에서 사라짐

```
public class StatelessService {
    public int order(String user, int price) {
        return price; // 지역변수로 처리
    }
}
```

2. 파라미터는 메소드 호출 시 전달된 값을 저장하며, 호출된 메소드 내부에서만 사용됨
3. threadlocal: 각 스레드마다 독립적인 변수를 제공함, 동일한 threadlocal객체라도 각 스레드는 자신만의 값을 가지며, 다른 스레드와 데이터를 공유하지 않음

```
public class ThreadLocalService {
    private ThreadLocal<Integer> threadLocalPrice =
```

```
    public void order(String user, int price) {
        threadLocalPrice.set(price); // 스레드별로 독립
    }
    public int getPrice() {
        return threadLocalPrice.get();
    }
}
```

❓ 13. 이 코드의 문제점이 뭘까?

▼ 정답

다른 2개의 MemoryMemberRepository가 생성되면서 싱글톤이 깨지는 것처럼 보인다

스프링 컨테이너는 어떻게 이 문제를 해결할까?

```
@Configuration
public class AppConfig {

    @Bean
    public MemberService memberService() {
        return new MemberServiceImpl(memberRepository());
    }

    @Bean
    public OrderService orderService() {
        return new OrderServiceImpl(
                memberRepository(),
                discountPolicy());
    }

    @Bean
    public MemberRepository memberRepository() {
        return new MemoryMemberRepository();
    }
    ...
}
```

❓ 14. 스프링 컨테이너는 어떻게 이 문제를 해결할까?

▼ 정답

스프링은 바이트코드를 조작하는 라이브러리를 사용함

```
@Test
void configurationDeep() {
    ApplicationContext ac = new
AnnotationConfigApplicationContext(AppConfig.class);

    //AppConfig도 스프링 빈으로 등록된다.
    AppConfig bean = ac.getBean(AppConfig.class);

    System.out.println("bean = " + bean.getClass());
    //출력: bean = class hello.core.AppConfig$$EnhancerBySpringCGLIB$$bd479d70
}
```

AnnotationConfigApplicationContext에 파라미터로 넘긴 값은 스프링 빈으로 등록된다

❓ 15. isEqualTo(),isSameAs(), isInstanceOf() 차이

▼ 정답

isEqualTo(),isSameAs(), isInstanceOf() 비교

isSameAs(): 주소값을 비교하는 메소드

isEqualTo(): 대상의 내용 자체를 비교하는 메소드

isInstanceOf(): 해당 타입의 인스턴스인지를 비교하는 메소드, 객체가 해당 클래스 혹은 그 하위 클래스로부터 생성된 객체면 true, 아니면 false를 반환

▼ 6 컴포넌트 스캔, 7 의존관계 자동 주입

스프링은 설정 정보가 없어도 자동으로 스프링 빈을 등록하는 컴포넌트 스캔 기능을 제공함

의존관계도 자동으로 주입하는 @Autowired 기능을 제공함

❓16. @Controller, @Repository, @Configuration의 역할

▼ 정답

@Controller: 스프링 MVC 컨트롤러로 인식

@Repository: 스프링 데이터 접근 계층으로 인식하고, 데이터 계층의 예외를 스프링 예외로 변환

@Configuration: 스프링 설정 정보로 인식하고, 스프링 빈이 싱글톤을 유지하도록 추가 처리함

❓17. 의존관계 주입 방법 4가지

▼ 정답

1. 생성자 주입: 생성자 호출 시점에 딱 1번만 호출되는것이 보장됨, 불변, 필수 의존관계에 사용함
2. 수정자 주입(setter 주입): 자바빈 프로퍼티 규약의 수정자 메소드 방식을 사용하는 방법

   https://velog.io/@hye9807/자바빈-규약
3. 필드 주입: 외부에서 변경이 불가능해서 테스트 하기 어려움, DI 프레임워크가 없으면 아무것도 할 수 없음

   ❓왜 DI프레임워크가 없으면 아무것도 할 수 없는걸까?

   ▼ 정답

   1. 스프링 같은 DI프레임워크가 객체를 생성하고 의존성을 주입해주는 역할을 담당함. 즉, 의존성 주입이 자동으로 이루어짐
   2. 객체 생성과 의존성 설정이 분리되지 않음. 객체를 생성한 후에 DI컨테이너가 내부적으로 리플렉션을 사용하여 필드에 접근하고 의존성을 주입함

   @Bean에서 파라미터에 의존관계는 자동 주입된다

```
@Bean
OrderService orderService(MemberRepository memberRepoisitory, DiscountPolicy
discountPolicy) {
    return new OrderServiceImpl(memberRepository, discountPolicy);
}
```

4. 일반 메소드 주입

롬복은 @RequiredArgsConstructor 기능을 제공하고, 이 기능을 사용하면 final이 붙은 필드와 @NotNull이 붙은 필드를 모아서 생성자를 자동으로 만들어준다

결론은 생성자 주입을 사용하자!

❓ 18. 그 이유는?

▼ 정답

1. 생성자 주입은 객체를 생성할 때 딱 1번만 호출되므로 이후에 호출되는 일이 없음 → 불변하게 설계
2. 생성자 주입을 사용하면 주입 데이터를 누락했을 때 컴파일 오류가 발생함 → 어떤 값을 주입해야할지 바로 알 수 있음
3. 생성자 주입을 사용하면 final 키워드를 사용할 수 있음 → 생성자에서 값이 설정되지 않는 오류를 컴파일 시점에 막아줌

```
@Component
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
```

```
    private final DiscountPolicy discountPolicy;

    @Autowired
    public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy
discountPolicy) {
        this.memberRepository = memberRepository;
    }
    //...
}
```

```
java: variable discountPolicy might not have been initialized
```

롬복이 자바의 애노테이션 프로세서 라는 기능을 이용해서 컴파일 시점에 생성자 코드를 자동으로 생성해준다

❓ 19. 조회 대상 빈이 2개 이상일때 어떤 어노테이션을 쓸까? (3개)

▼ 정답

@Autowired , @Qualifier, @Primary

1. @Autowired: 타입 매칭을 시도하고, 타입 매칭의 결과가 2개 이상일 때 필드명, 파라미터 명으로 빈 이름을 매칭한다
2. @Qualifier: 추가 구분자를 붙여주는 방법이다
3. @Primary: 우선순위를 정하는 방법

→ 메인 데이터베이스의 커넥션을 획득하는 스프링 빈은 @Primary를 적용해서 조회하는 곳에서 @Qualifier 지정 없이 편리하게 조회하고, 서브 데이터베이스 커넥션 빈을 획득할 때는 @Qualifier를 지정해서 획득

❓ 20. 어떤 경우에 컴포넌트 스캔과 자동 주입을 사용하고, 어떤 경우에 설정 정보를 통해서 수동으로 빈을 등록하고 의존관계도 수동으로 주입해야 할까?

스프링 부트는 컴포넌트 스캔을 기본으로 하고, 스프링 부트의 다양한 스프링 빈들도 조건이 맞으면 자동으로 등록하도록 설계

애플리케이션에 광범위하게 영향을 미치는 기술 지원 객체는 수동 빈으로 등록해서 설정 정보에 바로 나타나게 하는 것이 유지보수하기 좋음

▼ 8 빈 생명주기 콜백, 9 빈스코프

스프링 빈은 객체를 생성하고, 의존관계 주입이 다 끝난 다음에야 필요한 데이터를 사용할 수 있는 준비가 완료됨. 따라서 초기화 작업은 의존관계 주입이 모두 완료되고 난 다음에 호출해야 한다

스프링은 의존관계 주입이 완료되면 스프링 빈에게 콜백 메소드를 통해서 초기화 시점을 알려주는 다양한 기능을 제공함. 스프링은 스프링 컨테이너가 종료되기 직전에 소멸 콜백을 줌

스프링 빈의 이벤트 라이플사이클

스프링 컨테이너 생성 → 스프링 빈 생성 → 의존관계 주입 → 초기화 콜백 → 사용 → 소멸 전 콜백 → 스프링 종료

❓ 21. 스프링이 제공하는 빈 생명주기 콜백 방법 3가지

▼ 정답

1. 인터페이스 InitializingBean, DisposableBean

   InitializingBean는 afterProperties() 메소드로 초기화를 지원함

   DisposableBean는 destroy()메소드로 소멸 지원함

2. 설정 정보에 초기화 메소드, 종료 메소드 지정

   설정 정보에 @Bean(initMethod=”init”, destroyMethod=”close”) 처럼 초기화, 소멸 메소드를 지정

3. @PostConstruct, @PreDestroy 어노테이션 지원

→ @PostConstruct, @PreDestroy 어노테이션을 사용하고, 코드를 고칠수 없는 외부 라이브러리를 초기화, 종료해야 하면 @Bean의 initMethod, destroyMethod를 사용하자

❓ 22. 스프링 빈이 스프링 컨테이너의 시작과 함께 생성되어서 스프링 컨테이너가 종료될 때까지 유지됨. 그 이유는?

▼ 정답

스프링 빈이 싱글톤 스코프로 생성되기 때문이다

❓ 23. 스프링이 제공하는 다양한 스코프

▼ 정답

1. 싱글톤: 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위의 스코프
2. 프로토타입: 스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입까지만 관여함
3. 웹 관련 스코프: request, session, application

빈 스코프 등록 방법

1. 컴포넌트 스캔 자동 등록

```
@Scope("prototype")
@Component
public class HelloBean {}
```

2. 수동 등록

```
@Scope("prototype")
@Bean
PrototypeBean HelloBean() {
    return new HelloBean();
}
```

싱글톤 스코프의 빈을 조회하면 스프링 컨테이너는 항상 같은 인스턴스의 스프링 빈을 반환함

반면에 프로토타입 스코프를 스프링 컨테이너에 조회하면 스프링 컨테이너는 항상 새로운 인스턴스를 생성해서 반환함

싱글톤 빈은 스프링 컨테이너 생성 시점에 초기화 메소드가 실행되지만, 프로토타입 스코프의 빈은 스프링 컨테이너에서 빈을 조회할 때 생성되고, 초기화 메소드도 실행된다

프로토타입 빈 특징

스프링 컨테이너에 요청할 때마다 새로 생성

스프링 컨테이너는 프로토타입 빈의 생성과 의존관계 주입, 초기화까지만 관여

종료 메소드가 호출되지 않음

그래서 프로토타입 빈은 프로토타입 빈을 조회한 클라이언트가 관리해야 한다. 종료 메서드에 대한 호출도 클라이언트가 직접 해야함

❓ 24. 싱글톤 빈과 프로토타입 빈이 같이 사용의 문제점

▼ 정답

싱글톤 빈은 생성 시점에만 의존관계 주입을 받기 때문에, 프로토타입 빈이 새로 생성되기는 하지만, 싱글톤 빈과 함께 계속 유지되는 것이 문제임

사용할때마다 새로 생성되는것을 원함

❓ 25. 프로토타입 스코프, 싱글톤 빈과 함께 사용시 어떻게 하면 사용할 때마다 새로운 프로토타입 빈을 생성할 수 있을까?

▼ 정답

1. 싱글톤 빈이 프로토타입을 사용할때마다 스프링 컨테이너에 새로 요청

핵심 코드

```
@Autowired
private ApplicationContext ac;
```

```
public int logic() {
    PrototypeBean prototypeBean = ac.getBean(PrototypeBean.class);
    prototypeBean.addCount();
    int count = prototypeBean.getCount();
    return count;
}
```

직접 필요한 의존관계를 찾는다 = DL(dependency lookup)

그런데, 이렇게 스프링의 애플리케이션 컨텍스트 전체를 주입받게 되면, 스프링 컨테이너에 종속적인 코드가 되고 단위 테스트도 어렵다

2. ObjectFactory, ObjectProvider

```
@Autowired
private ObjectProvider<PrototypeBean> prototypeBeanProvider;

public int logic() {
    PrototypeBean prototypeBean = prototypeBeanProvider.getObject();
    prototypeBean.addCount();
    int count = prototypeBean.getCount();
    return count;
}
```

ObjectFactory는 기능이 단순, 별도의 라이브러리 필요 없음, 스프링에 의존

ObjectProvider는 ObjectFactory상속, 옵션, 스트림 처리 등 편의 기능이 많고, 별도의 라이브러리 필요 없음

❓ 26. 프로토타입 빈을 언제 사용할까?

▼ 정답

매번 사용할때마다 의존관계 주입이 완료된 새로운 객체가 필요하면 사용한다

웹스코프

웹 환경에서만 동작하고, 프로토타입과 다르게 스프링이 해당 스코프의 종료시점까지 관리함. 따라서 종료 메서드가 호출됨

request, session, application, websocket 스코프가 존재함

spring-boot-starter-web 라이브러리를 추가하면 스프링 부트는 내장 톰켓 서버를 활용해서 웹 서버와 스프링을 함께 실행한다

스프링 부트는 웹 라이브러리가 없으면 annotationConfigApplicationContext을 기반으로 애플리케이션을 구동한다.

request스코프를 쓰면 동시에 여러 http 요청이 올때 어떤 요청이 남긴 로그인지 구분가능하다

UUID를 사용해서 HTTP 요청을 구분하자, request URL 정보도 넣어서 어떤 URL을 요청해서 남긴 로그인지 확인하자

▼ MyLogger

```
@Component
@Scope(value = "request") // request 스코프 지정, http 요청당
 public class MyLogger {
     private String uuid;
     private String requestURL;
     public void setRequestURL(String requestURL) {
         this.requestURL = requestURL;
}
     public void log(String message) {
         System.out.println("[" + uuid + "]" + "[" + reque
         message); }
    @PostConstruct // 빈이 생성되는 시점에 @PostConstruct초기화
    public void init() {
        uuid = UUID.randomUUID().toString();
        System.out.println("[" + uuid + "] request scope b
    }
    @PreDestroy // 빈이 소멸되는 시점에 @PreDestroy를 사용해서 종
    public void close() {
```

```
        System.out.println("[" + uuid + "] request scope b
    }
}
```

▼ LogDemoController

```
@Controller
@RequiredArgsConstructor
public class LogDemoController {
    private final LogDemoService logDemoService;
    private final MyLogger myLogger;
    @RequestMapping("log-demo")
    @ResponseBody
    public String logDemo(HttpServletRequest request) {
        String requestURL = request.getRequestURL().toStri
        myLogger.setRequestURL(requestURL);
        myLogger.log("controller test");
        logDemoService.logic("testId");
        return "OK";
    }
}
```

로거가 잘 작동하는지 확인

▼ LogDemoService

```
 @Service
 @RequiredArgsConstructor
 public class LogDemoService {
 private final MyLogger myLogger;
     public void logic(String id) {
         myLogger.log("service id = " + id);
    }
}
```

기대하는 출력

```
[d06b992f...] request scope bean create
[d06b992f...][http://localhost:8080/log-demo] controller test
[d06b992f...][http://localhost:8080/log-demo] service id = testId
[d06b992f...] request scope bean close
```

실제는 기대와 다르게 애플리케이션 실행 시점에 오류 발생

```
Error creating bean with name 'myLogger': Scope 'request' is not active for the
current thread; consider defining a scoped proxy for this bean if you intend to
refer to it from a singleton;
```

스프링 애플리케이션을 실행 시키면 오류가 발생함

스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입이 가능하지만, request 스코프 빈은 아직 생성되지 않는다

해결 방안들

1. 스코프와 provider

   LogDemoController

```
@Controller
@RequiredArgsConstructor
public class LogDemoController {
   private final LogDemoService logDemoService;
   private final ObjectProvider<MyLogger> myLoggerProvider
   @RequestMapping("log-demo")
   @ResponseBody
   public String logDemo(HttpServletRequest request) {
       String requestURL = request.getRequestURL().toStri
       MyLogger myLogger = myLoggerProvider.getObject();
       myLogger.setRequestURL(requestURL);
       myLogger.log("controller test");
       logDemoService.logic("testId");
       return "OK";
```

```
 }
}
```

LogDemoService

```
@Service
@RequiredArgsConstructor
public class LogDemoService {
   private final ObjectProvider<MyLogger> myLoggerProvider
   public void logic(String id) {
       MyLogger myLogger = myLoggerProvider.getObject();
       myLogger.log("service id = " + id);
}
}
```

ObjectProvider 덕분에 ObjectProvider.getObject()를 호출하는 시점까지 request scope 빈의 생성을 지연할 수 있음

2. 스코프와 프록시

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyLogger {
}
```

MyLogger의 가짜 프록시 클래스를 만들어두고 http request와 상관없이 가짜 프록시 클래스를 다른 빈에 미리 주입해둘 수 있음

LogDemoController

```
@Controller
@RequiredArgsConstructor
public class LogDemoController {
   private final LogDemoService logDemoService;
   private final MyLogger myLogger;
   @RequestMapping("log-demo")
   @ResponseBody
```

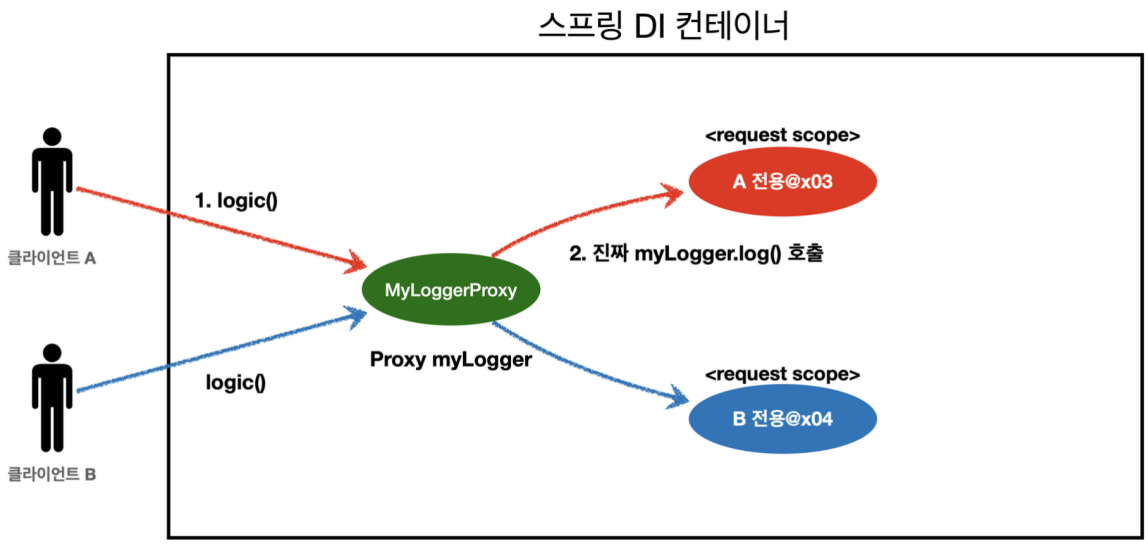
```
    public String logDemo(HttpServletRequest request) {
        String requestURL = request.getRequestURL().toStri
        myLogger.setRequestURL(requestURL);
        myLogger.log("controller test");
        logDemoService.logic("testId");
        return "OK";
 }
}
```

LogDemoService

```
@Service
@RequiredArgsConstructor
public class LogDemoService {
   private final MyLogger myLogger;
   public void logic(String id) {
        myLogger.log("service id = " + id);
 }
}
```

CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만들어서 주입한다

`@Scope` 의 `proxyMode = ScopedProxyMode.TARGET_CLASS)` 를 설정하면 스프링 컨테이너는 CGLIB
라는 바이트코드를 조작하는 라이브러리를 사용해서, MyLogger를 상속받은 가짜 프록시 객체를 생성한다.



가짜 프록시 객체는 요청이 오면 내부에서 진짜 빈을 요청하는 위임 로직이 들어있음

가짜 프록시 객체는 내부에 진짜 myLogger를 찾는 방법을 알고 있음, 클라이언트가 myLogger.log()을 호출하면 가짜 프록시 객체의 메소드를 호출한 것

가짜 프록시 객체는 request스코프의 진짜 myLogger.log()를 호출함

가짜 프록시 객체는 원본 클래스를 상속 받아서 만들어졌기 때문에 이 객체를 사용하는 클라이언트 입장에서는 원본인지 아닌지 모르게 동일하게 사용함