단축키

생성자(Generate): Ctrl + N

에러 수정: Arlt + Enter

스마트 완성: Ctrl + Shift + Enter

Extract / Introduce : Ctrl + Art + V

오류난곳 확인: F2

테스트 생성: Ctrl + Shift + t

히스토리 보기: Ctrl + e

바로 밑에 줄로 이동하기: Ctrl + Shift + Enter

클래스로 이동 : Ctrl + o

코드 정리: Ctrl + Alt + Shift + L

인라인 정리: Ctrl + Alt + N

내용 정리

클라이언트 코드의 의존관계 수정

- 클라이언트 코드인 (OrderServiceImpl) 은 (DiscountPolicy)의 인터페이스 뿐만 아니라 구체 클래스도 의존하고 있음

  DIP 위반 - 추상에만 의존하도록 변경(인터페이스에만 의존)

변경 전

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

변경 후

```
public class OrderServiceImpl implements OrderService{
    private DiscountPolicy discountPolicy;
}
```

구현 객체의 생성을 다른곳에 맡기도록 한다.

관심사의 분리

- **배우**는 본인의 역할인 배역을 수행하는 것에만 집중하면 된다.
- **공연 기획자**는 공연을 구성하고, 배우를 섭외하고, 역할에 맞는 배우를 지정하는것에 집중하면 된다.

AppConfig의 활용

- 애플리케이션의 전체 동작 방식을 구성(config)하기 위해, **구현 객체를 생성**하고, **연결**하는 클래스

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

- 실제 동작에 필요한 **구현 객체를 생성**한다. --> OrderServiceImpl, MemoryMemberRepository, FixDiscountPolicy
- 생성한 객체 인스턴스의 참조(레퍼런스)를 **생성자를 통해서 인젝션 (주입(연결))**해준다.

  OrderServiceImpl --> MemoryMemberRepository, FixDiscountPolicy

변경 전 OrderServiceImpl

```
//변경전 OrderServiceImpl
public class OrderServiceImpl implements OrderService {

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

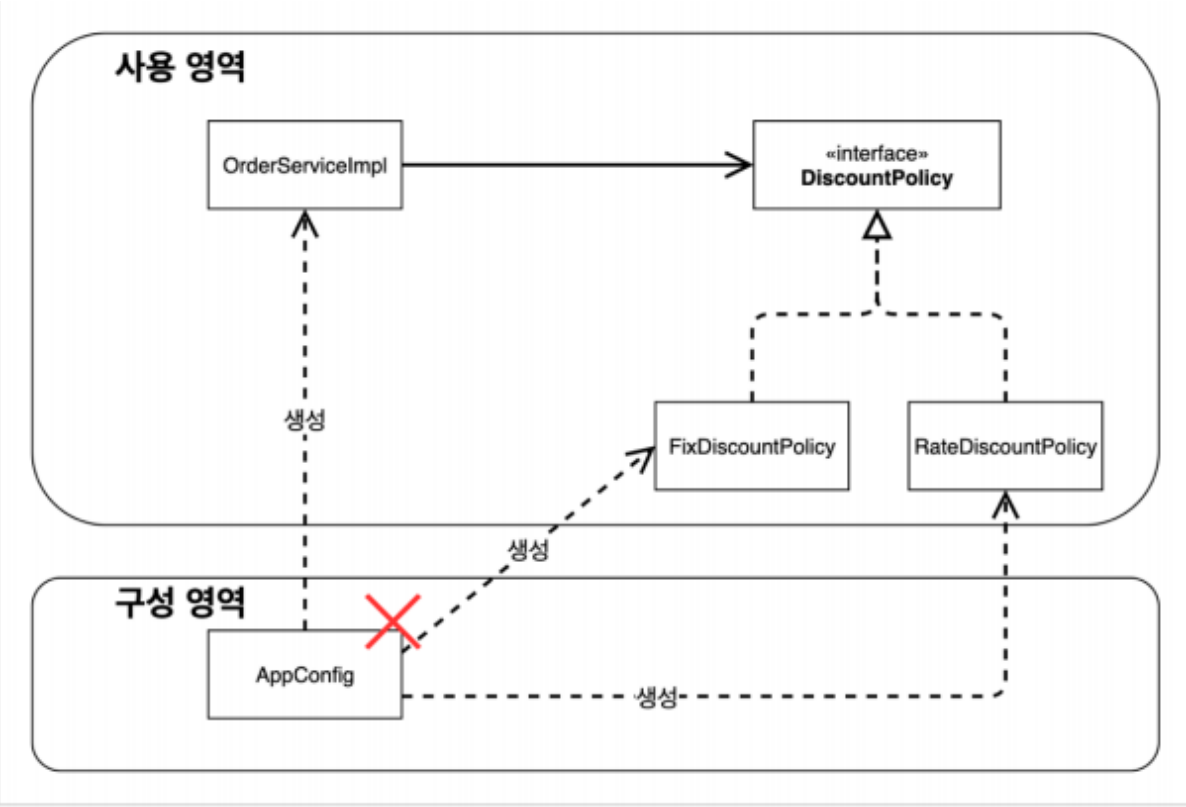
변경 후 OrderServiceImpl

```
//변경된 OrderServiceImpl
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;

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

- 설계 변경으로 OrderServiceImpl은 FixDiscountPolicy를 의존하지 않는다
- 단지 DiscountPolicy 인터페이스만 의존한다.
- OrderServiceImpl 입장에서 생성자를 통해 어떤 구현 객체가 주입될지는 알 수 없다.
- OrderServiceImpl의 생성자를 통해 어떤 구현 객체를 주입할지는 외부(AppConfig) 에서 결정한다.
- OrderServiceImpl는 이제 실행에만 집중하면 된다.

AppConfig의 등장으로 애플리케이션이 크게 사용 영역과, 객체를 생성하고 구성하는 영역으로 분리되었다.

- 이점: FixDiscountPolicy를 RateDiscountPolicy로 변경해도 구성 영역만 영향을 받고, 사용 영역은 영향 받지 않는다.

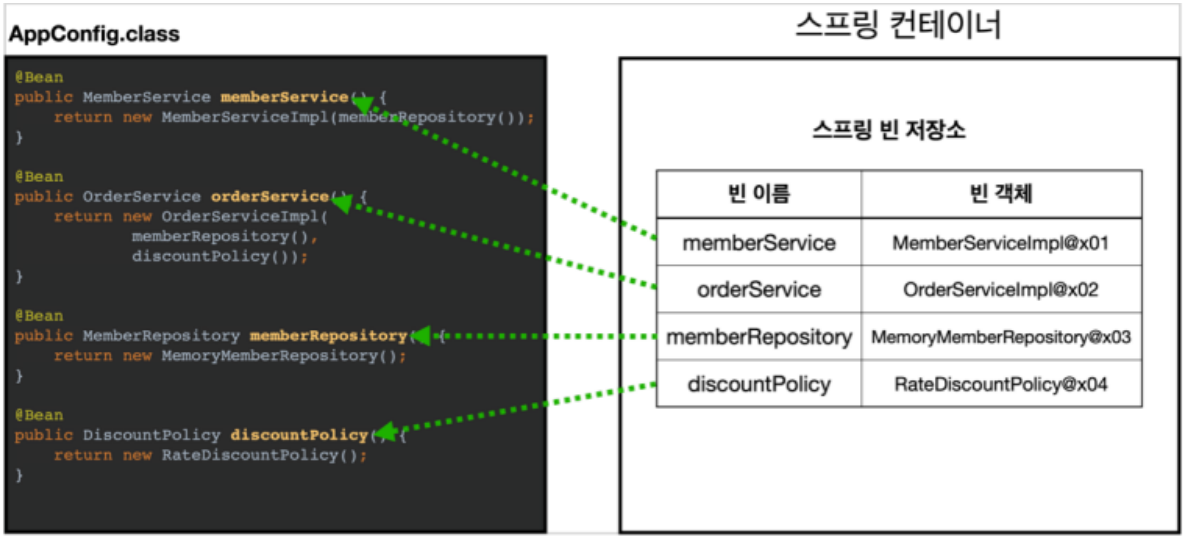


스프링 컨테이너와 스프링 빈

스프링 컨테이너 생성

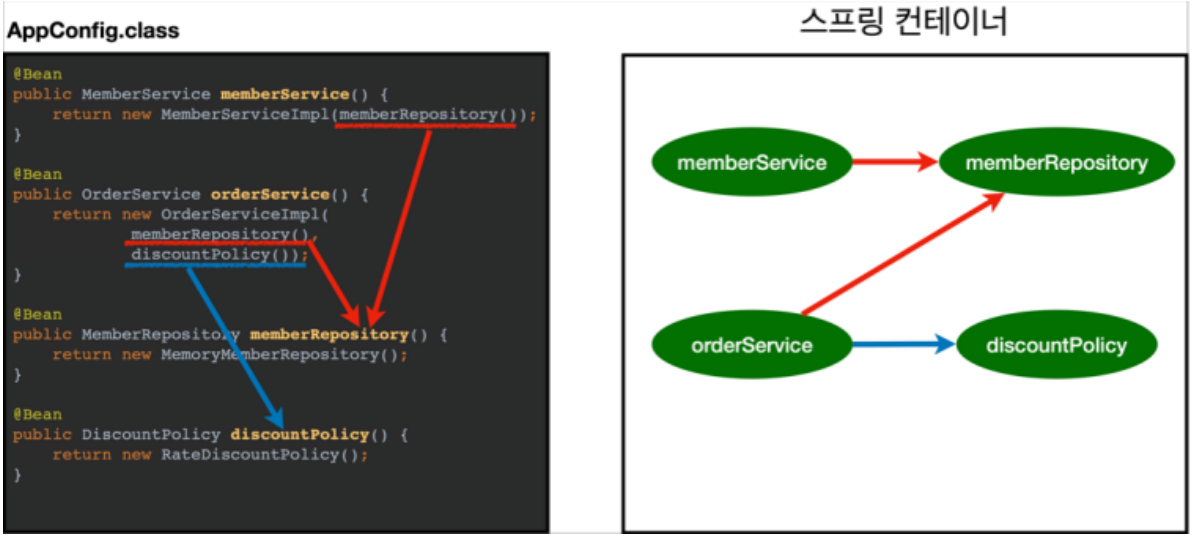
```
//스프링 컨테이너 생성
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AppConfig.class);
```

- ApplicationContext 를 **스프링 컨테이너**라고 한다.
- AnnotationConfigApplicationContext 는 ApplicationContex **인터페이스의 구현체**이다.
- 스프링 컨테이너 생성시에는 구성 정보 지정이 필요하다. ----> Appconfig.class 를 구성정보로 활용



- 파라미터로 넘어온 설정 클래스 정보(Appconfig.class) 를 사용해서 스프링 빈을 등록한다.
- 빈 이름

    - 메서드 이름을 사용한다.
    - 직접 부여할 수 있다. @Bean(name = "memberService2") *** 빈 이름은 항상 다른 이름을 부여해야 한다.**
- 스프링 빈 의존 관계 설정

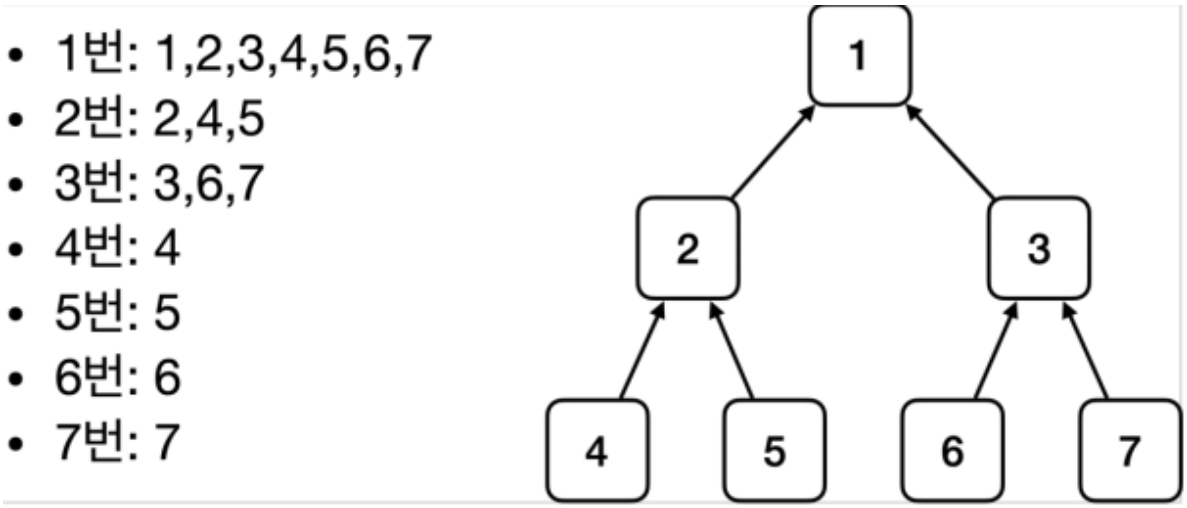


스프링 빈 조회 - 기본

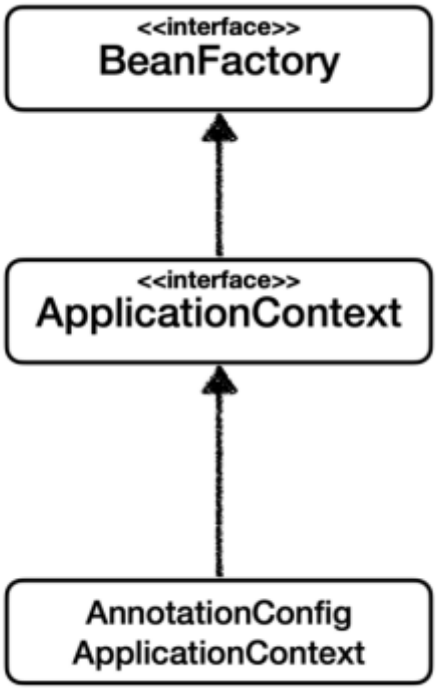
- ac.getBean(빈이름, 타입)
- ac.getBean(타입)
- 조회 대상 스프링 빈이 없으면 예외발생(NoSuchBeanDefinitionException)
- 타입으로 조회시 같은 타입의 스프링 빈이 둘 이상이면 오류가 발생한다 ---> 빈 이름을 지정
- ac.getBeanOfType()을 사용하여 해당 타입의 모든 빈 조회 가능

스프링 빈 조회 - 상속관계

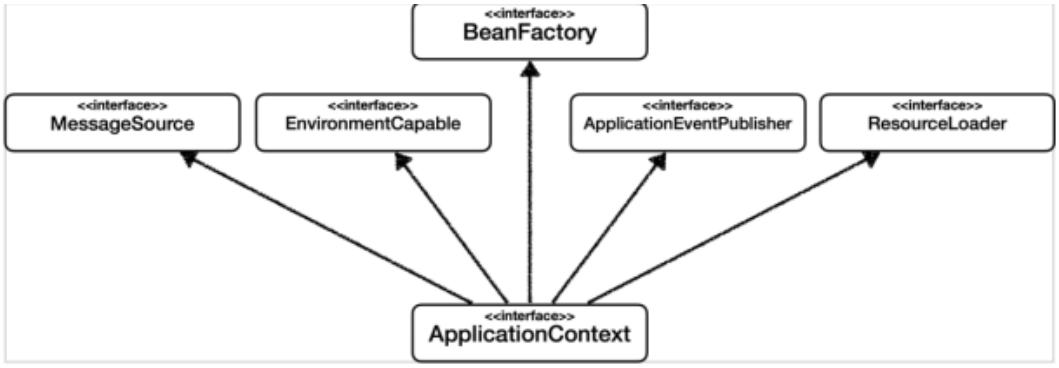
- 부모 타입으로 조회하면, 자식 타입도 함께 조회한다.
- 모든 자바 객체의 최고 부모인 Object 타입으로 조회하면, 모든 스프링 빈을 조회한다.



- 구성도



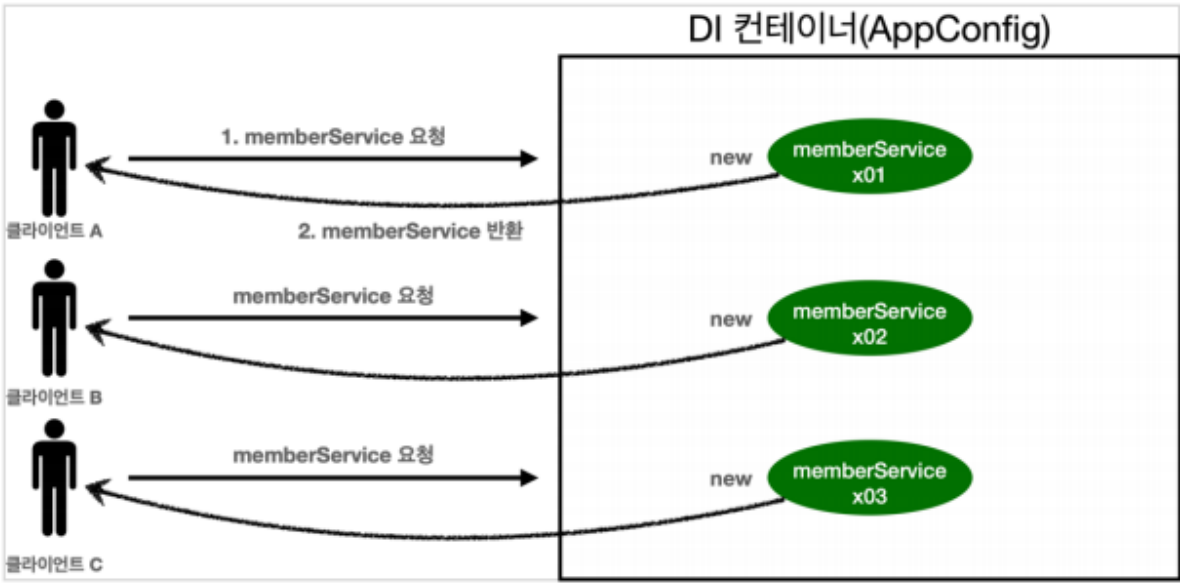
- **BeanFactory**
  - 스프링 컨테이너의 최상위 인터페이스
  - 스프링 빈을 관리하고 조회하는 역할을 담당
  - getBean()을 제공
- **ApplicationContext**
  - BeanFactory의 기능을 모두 상속받아 제공
  - 메시지 소스를 활용한 국제화 기능 / 환경변수/ 애플리케이션 이벤트/ 편리한 리소스 조회 부가기능 제공



싱글톤 컨테이너

싱글톤의 필요성

- 대부분의 스프링 에플리케이션은 웹 어플리케이션이다. ---> 웹 어플리케이션은 보통 여러 고객이 **동시에 요청**한다.



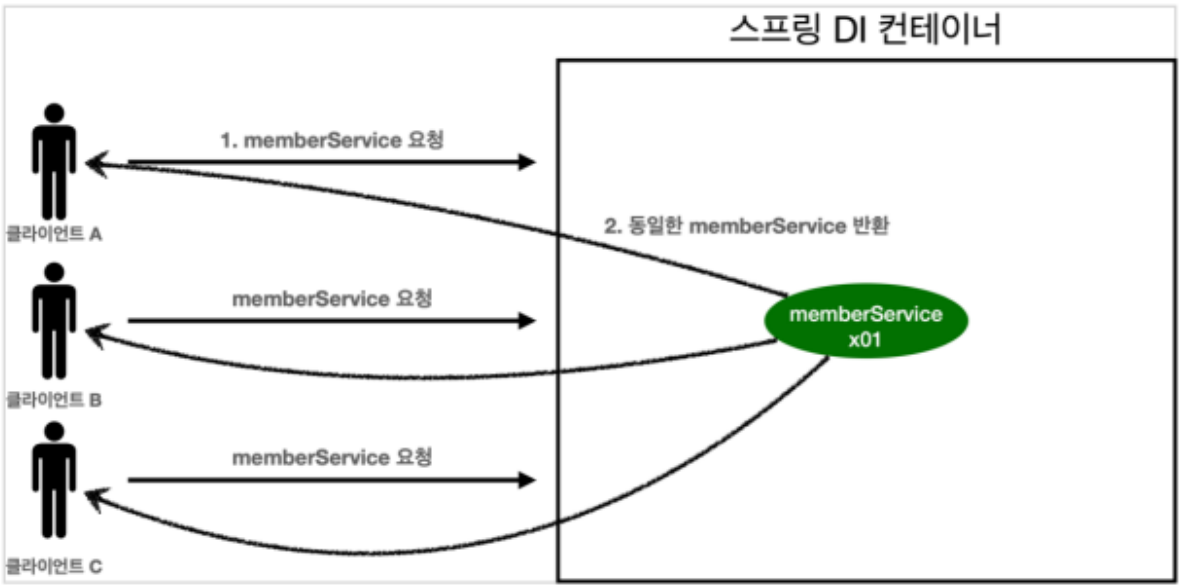
- 스프링 없는 순수한 DI 컨테이너라면 초당 100의 트레픽이면 100개의 객체가 생성된다. -> **메모리 낭비**
- 해당 객체가 딱 1개만 생성되고, 공유하도록 설계하면 해결된다. -> **싱글톤 패턴**

싱글톤 패턴

- 클래스의 인스턴스가 딱 1개만 생성되는 것을 보장하는 디자인 패턴이다.
- private 생성자를 사용해서 외부에서 임의로 new 키워드를 사용하지 못하도록 막아야 한다.
    1. static 영역에 객체 instance를 미리 하나 생성해서 올려둔다.
    2. 이 객체 인스턴스가 필요하면 오직 getInstance() 메서드를 통해서만 조회할 수 있다. 이 메서드를
       호출하면 항상 같은 인스턴스를 반환한다.
    3. 딱 1개의 객체 인스턴스만 존재해야 하므로, 생성자를 private으로 막아서 혹시라도 외부에서 new
       키워드로 객체 인스턴스가 생성되는 것을 막는다.

싱글톤 컨테이너

- 싱글톤 패턴의 여러 문제를 해결하면서, **스프링컨테이너가 싱글톤으로 관리**해준다.
- 스프링 컨테이너는 싱글톤 컨테이너 역할을 한다 싱글톤 객체를 생성하고 관리하는 기능 : **싱글톤 레지스트리**



- 고객의 요청이 올 때 마다 객체를 생성하는 것이 아니라, 이미 만들어진 객체를 공유해서 효율적으로 재사용 가능

싱글톤 방식의 주의점

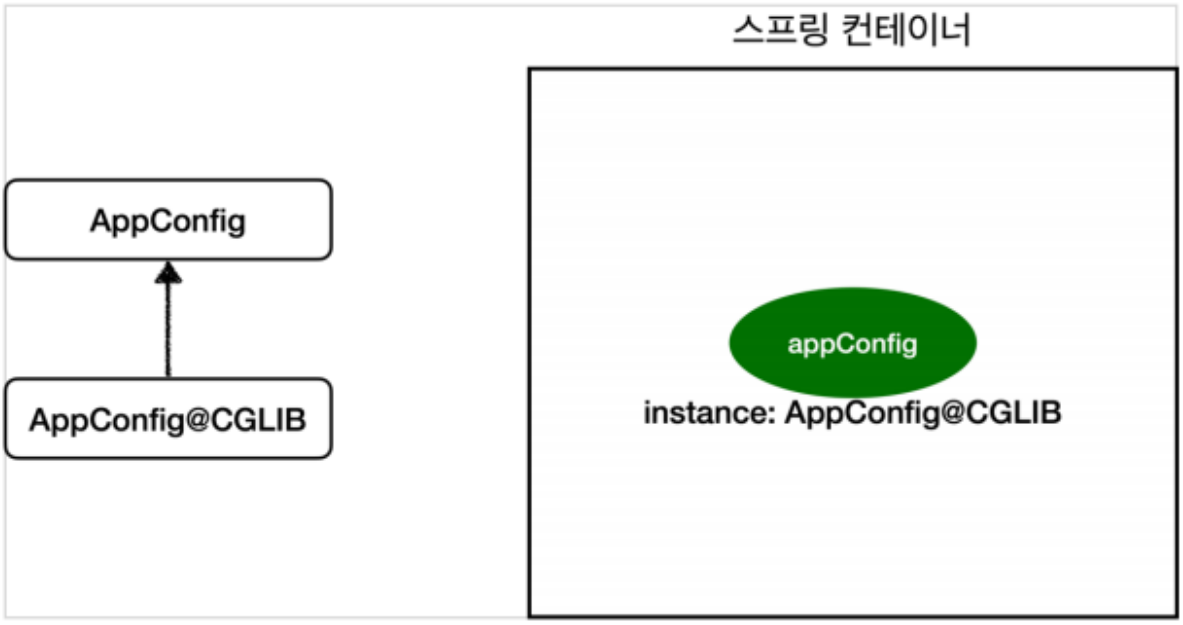
- 객체 인스턴스를 하나만 생성해서 공유하기 때문에 싱글톤 객체는 상태를 유지(Stateful) 하게 설계해서는 안된다.
- **무상태(Stateless)로 설계** 해야 한다.
  - 특정 클라이언트에 의존적인 필드가 있으면 안된다.
  - 특정 클라이언트가 값을 변경할 수 있는 필드가 있으면 안된다.
  - 가급적 읽기만 가능해야 한다.
  - 필드 대신에 자바에서 공유되지 않는 지역변수, 파라미터, ThreadLocal 등을 사용해야 한다.
- 스프링 빈의 필드에 **공유 값을 설정**하면 **큰 장애**가 발생할 수 있다!
  - 예시) ThreadA가 사용자A 코드를 호출하고 ThreadB가 사용자B 코드를 호출한다고 가정
  - price 필드는 공유되는 필드인데 특정 클라이언트가 값을 변경하면 10000원의 주문금액 대신 20000원이 된다.

@Configuration 과 싱글톤

- @Configuration 은 등록된 Bean이 싱글톤을 보장하게 해준다.
- 스프링 컨테이너는 싱글톤 레지스트리다. 따라서 빈이 싱글톤이 되도록 보장해주어야 한다.
- 스프링이 자바 코드까지 컨트롤할 수 없다 --> **클래스의 바이트코드를 조작하는 라이브러리를 사용**

```
@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
}
```

- AnnotationConfigApplicationContext 에 파라미터로 넘긴 값은 스프링 빈으로 등록된다 ---> AppConfig도 스프링 빈
- class hello.core.AppConfig 대신 다른 값이 출력이 된다.
- 스프링이 CGLIB라는 바이트코드 조작 라이브러리를 사용해서 AppConfig클래스를 상속받은 **임의의 다른 클래스**를 만들고, **다른 클래스를 스프링 빈으로 등록한 것이다**



- @Bean이 붙은 메서드마다 이미 스프링 **빈이 존재하면 존재하는 빈을 반환**
- **스프링 빈이 없으면 생성해서 스프링 빈으로 등록하고 반환**하는 코드가 동작

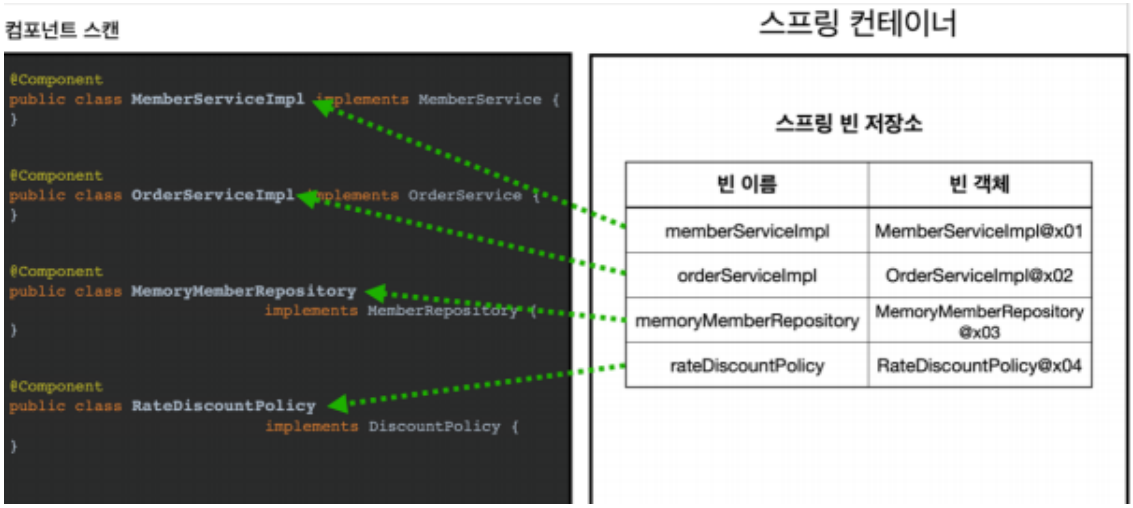
컴포넌트 스캔

컴포넌트 스캔과 의존관계 자동 주입

- 등록해야 할 스프링 빈이 많아지면 문제가 생길 가능성이 있다.
- 스프링은 설정 정보가 없어도 **자동으로 스프링 빈을 등록하는 컴포넌트 스캔**이라는 기능을 제공한다.
- **의존관계도 자동으로 주입하는 @Autowired**라는 기능도 제공한다.
- @ComponentScan을 설정 정보에 붙여주면 된다. --> 클래스에 @Component 를 붙여주면 된다.

```
@Component
public class MemberServiceImpl implements MemberService {

    private final MemberRepository memberRepository;

    @Autowired
    public MemberServiceImpl(MemberRepository memberRepository){
        this.memberRepository = memberRepository;
    }
}
```

1. @ComponentScan



- @ComponentScan은 @Componet 가 붙은 모든 클래스를 스프링 빈으로 등록한다.
- 클래스명을 사용하되 맨 앞글자만 소문자를 사용한다.
  - MemberServiceImpl 클래스 -> memberServiceImpl

2. @Autowired 의존관계 자동 주입
   - 생성자가 딱 1개만 있으면 @Autowired를 생략할 수 있다.

- 생성자에 @Autowired를 지정하면, 스프링 컨테이너가 자동으로 해당 스프링 빈을 찾아서 주입한다.
- 이때 기본 조회 전략은 타입이 같은 빈을 찾아서 주입한다.

- 생성자에 파라미터가 많아도 다 찾아서 자동으로 주입한다.

탐색 위치와 기본 스캔 대상

- 프로젝트 시작 루트에 AppConfig 같은 메인 설정 정보를 두고, @ComponetScan 애노테이션을 붙인다.

중복 등록과 충돌

1. 자동 빈 등록 vs 자동 빈 등록

   컴포넌트 스캔에 의해 자동으로 스프링 빈이 등록되는데, **그 이름이 같은 경우 스프링은 오류를 발생**시킨다.

   -> ConflictingBeanDefinitionException 예외 발생

2. 수동 빈 등록 vs 자동 빈 등록

   수동 빈 등록이 우선권을 가진다(수동 빈이 자동 빈을 오버라이딩 해버린다.)

수동 빈 등록시 남는 로그

```
Overriding bean definition for bean 'memoryMemberRepository' with a different
definition: replacing
```

최근 스프링 부트는 수동 빈 등록과 자동 빈 등록이 충돌나면 오류가 발생하도록 기본 값을 변경했다.

수동 빈 등록, 자동 빈 등록 오류시 스프링 부트 에러

```
Consider renaming one of the beans or enabling overriding by setting
spring.main.allow-bean-definition-overriding=true
```

의존관계 자동 주입

다양한 의존관계 주입방법

- **생성자 주입(주로 사용!!!)**

```
@Component
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;

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

- 수정자 주입(setter 주입)
- 필드 주입
- 일반 메서드 주입

생성자 주입을 사용하는 이유

- **불변**
  - 대부분의 의존관계 주입은 한번 일어나면 애플리케이션 종료시점까지 의존관계를 변경할 일이 없다.
  - 수정자 주입 : setXxx 메서드를 public으로 열어두어야 한다. --> 문제가 발생할 수 있다.
- **누락 방지**
  - 의존관계 주입이 누락이 되는 경우를 막아줄 수 있다.
- **final 키워드 사용 가능**
  - 필드에 final 키워드를 사용할 수 있다. ---> 생성자에 혹시 값이 설정되지 않는 오류를 막아준다.

롬복과 최신 트렌드

- 롬복 라이브러리가 제공하는 **@RequiredArgsConstructor** 기능을 사용하면 final이 붙은 필드를 모아서

  생성자를 자동으로 만들어준다.

```
@Component
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;
}
```

- 라이브러리 적용 (build.gradle)

```
//build.gralde
//lombok 설정 추가 시작
configurations {
     compileOnly {
         extendsFrom annotationProcessor
    }
}
//lombok 설정 추가 끝
```

조회 빈이 2개 이상 -문제

- @Autowired는 타입(Type)으로 조회한다.
- 같은 타입의 빈이 2개이상 조회가 되면 오류가 발생한다.

조회 빈이 2개 이상 - 해결방법 :@Autowired 필드 명, @Qulifier, @Primary

- **@Autowired 필드 명 매칭**
  - 타입 매칭 결과가 2개 이상일 때 필드 명, 파라미터 명으로 빈 이름 매칭
  - 예시) private DiscountPolicy discountPolicy --> private DiscountPolicy rateDiscountPolicy
- **@Qualifier -> @Qualifier 끼리 매칭 -> 빈 이름 매칭**
  - 추가 구분자를 붙여주는 방식
  - 예시) @Qualifier("rateDiscountPolicy"), @Qualifier("mainDiscountPolicy") 로 표기
- **@Primary 사용**

    - 우선 순위를 정하는 방법 @Primary가 우선권을 가진다.
    - **메인 데이터베이스의 커넥션**을 획득하는 스프링 빈은 **@Primary**를 적용하고
    - **서브 데이터베이스 커넥션** 스프링 빈을 획들할 때는 **@Qualifier**를 지정해서 명시적으로 활용

자동, 수동 올바른 실무 운영 기준

- 대부분 **자동 기능**을 기본으로 사용하자
    - **업무 로직 빈**: 웹을 지원하는 컨트롤러, 핵심 비지니스 로직이 있는 서비스, 데이터 계층의 로직을 처리하는 리포지토리등이 모두 업무 로직이다. 보통 비지니스 요구사항을 개발할 때 추가되거나 변경된다.
- 직접 등록하는 기술 지원 객체는 **수동 등록**
    - **기술 지원 빈**: 기술적인 문제나 공통관심사(AOP)를 처리할 때 주로 사용된다. 데이터베이스 연결이나 공통 로그 처리 처럼 업무 로직을 지원하기 위한 하부 기술이나 공통 기술
    - **다형성을 적극 활용 할때**: 등록 된 빈을 한눈에 알아보게 / **자동으로 사용한다면, 특정 패키지에 같이 묶어두자**

```
@Configuration
public class DiscountPolicyConfig {
    @Bean
    public DiscountPolicy rateDiscountPolicy() {
        return new RateDiscountPolicy();
    }

    @Bean
    public DiscountPolicy fixDiscountPolicy() {
        return new FixDiscountPolicy();
    }
}
```

빈 생명주기 콜백

빈 생명주기 콜백 시작

- 데이터베이스 커넥션 풀이나, 네트워크 소켓처럼 애플리케이션 시작 시점에 필요한 연결을 미리 해두고, 애플리케이션 종료 시점에 연결을 모두 종료하는 작업을 진행하려면, **객체의 초기화**와 **종료 작업**이 필요하다.
- 스프링 빈은 **객체 생성** ----> **의존관계 주입** 라이프 사이클을 가진다.
- 객체를 생성하고 의존관계 주입이 다 끝난 다음에야 데이터를 사용할 수 있는 준비가 된다.
- 스프링은 **의존관계 주입이 완료되면 스프링 빈에게 콜백 메서드**를 통해 초기화 시점을 알려준다.

스프링 빈의 이벤트 라이프 사이클

- **스프링 컨테이너 생성 --> 스프링 빈 생성 --> 의존관계 주입 --> 초기화 콜백 --> 사용 --> 소멸전 콜백 --> 스프링 종료**
- 초기화 콜백: 빈이 생성되고, 빈의 의존관계 주입이 완료된 후 호출
- 소멸전 콜백: 빈이 소멸되기 직전에 호출

빈 등록 초기화, 소멸 메서드 지정

- 설정 정보에 @Bean(initMethod = "init", destroyMethod = "close") 처럼 초기화, 소멸 메서드를 지정할 수 있다.

```
@Configuration
static class LifeCycleConfig {

    @Bean(initMethod = "init", destroyMethod = "close")
    public NetworkClient networkClient() {
        NetworkClient networkClient = new NetworkClient();
        networkClient.setUrl("http://hello-spring.dev");
        return networkClient;
    }
}
```

```
public void init() {
    System.out.println("NetworkClient.init");
    connect();
    call("초기화 연결 메시지");
}

public void close() {
    System.out.println("NetworkClient.close");
    disConnect();
}
```

- 메서드 이름을 자유롭게 줄 수 있다.
- 스프링 빈이 스프링 코드에 의존하지 않는다.
- 코드가 아니라 설정 정보를 사용하기 때문에 외부 라이브리에도 초기화, 종료 메서드를 적용할 수 있다.

애노테이션 @PostConstruct, @PreDestroy

- 최신 스프링에서 가장 권장하는 방법

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@PostConstruct
public void init() {
    System.out.println("NetworkClient.init");
    connect();
    call("초기화 연결 메시지");
}

@PreDestroy
public void close() {
    System.out.println("NetworkClient.close");
    disConnect();
}
```

- 스프링에 종속적인 기술이 아닌 JSR-250 자바 표준이므로 다른 컨테이너에서도 동작한다.
- 컴포넌트 스캔과 잘 어울린다.
- 외부 라이브러리에서는 적용하지 못한다.---> @Bean 의 initMethod, destroyMethod를 사용하자!

빈 스코프

빈 스코프란?

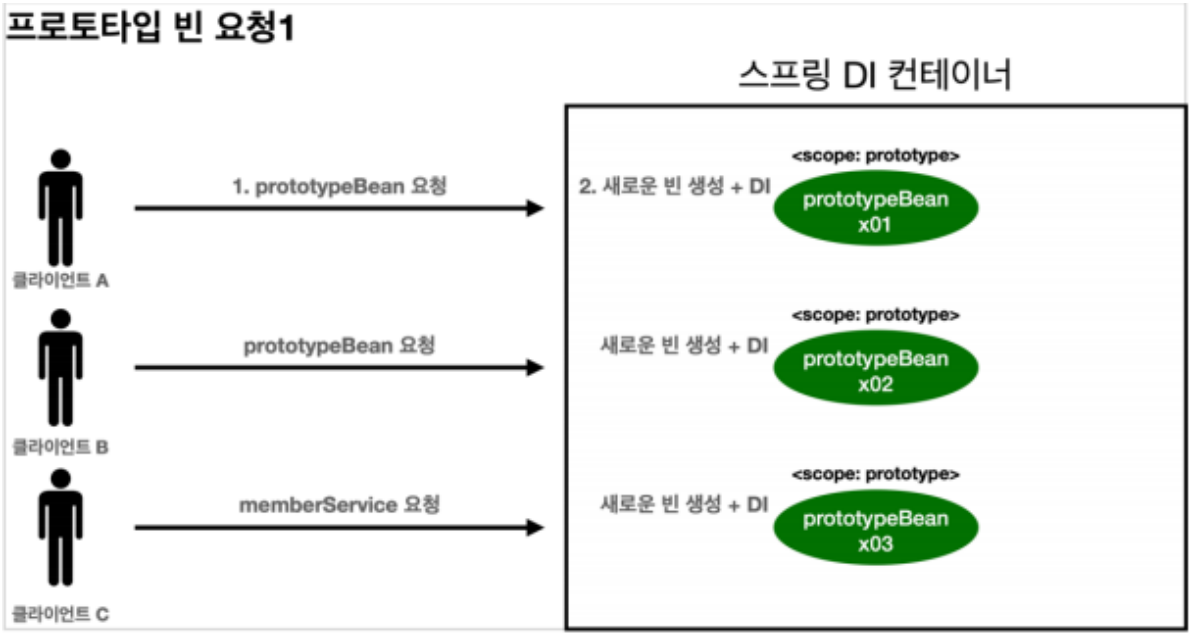
- 스프링 빈은 스프링 컨테이너의 시작과 함께 생성되어 스프링 컨테이너가 종료될 때까지 유지된다.
- 스프링 빈이 기본적으로 싱글톤 스코프로 생성되기 때문이다.
- 스코프란 빈이 존재할 수 있는 범위를 뜻한다.

스프링의 스코프 종류

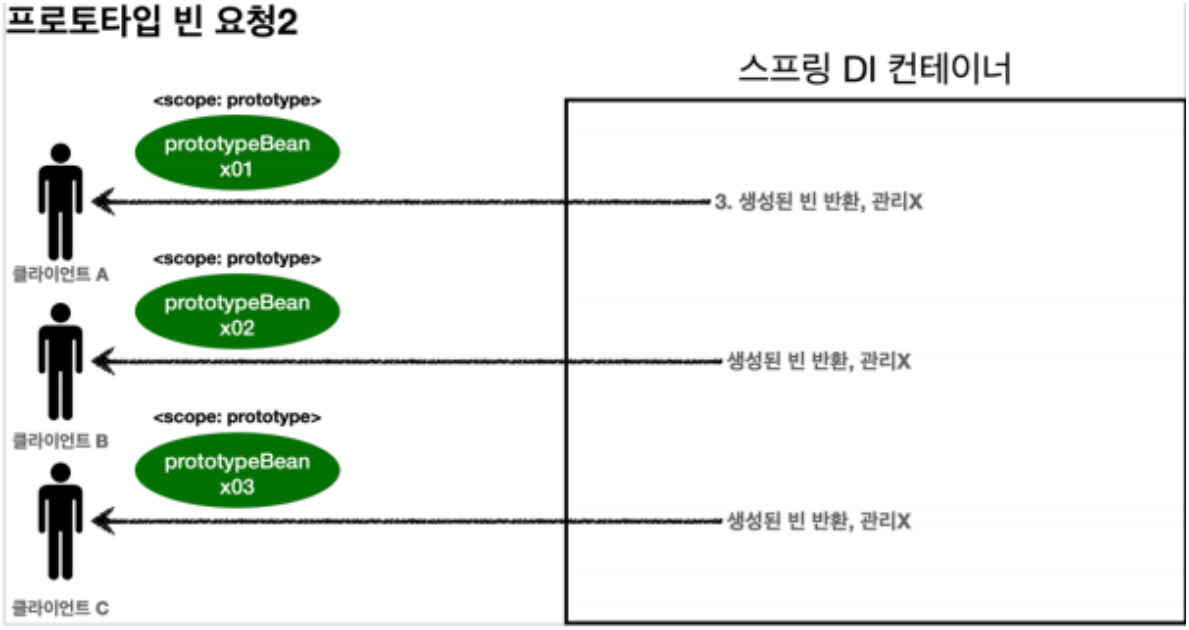
- **싱글톤**: 기본 스코프, 스프링 컨테이너의 시작과 종료까지 유지되는 가장 넓은 범위의 스코프
- **프로토타입**: 프로토타입 빈의 생성과 의존관계 주입까지만 관여하고 더는 관리하지 않는 매우 짧은 범위의 스코프
- **웹 관련 스코프**
  - **request**: 웹 요청이 들어오고 나갈때 까지 유지되는 스코프
  - **session**: 웹 세션이 생성되고 종료될 때 까지 유지되는 스코프
  - **application**: 웹의 서블릿 컨텍스트와 같은 범위로 유지되는 스코프

프로토타입 스코프

- 매번 사용할 때마다 의존관계 주입이 완료된 **새로운 객체가 필요할때** 사용하면 된다.
- 실제로는 싱글톤 빈으로 대부분 문제가 해결되기 때문에 사용하는 경우는 **극히 드물다**



1. 프로토타입 스코프의 빈을 스프링 컨테이너에 요청한다.
2. 스프링 컨테이너는 이 시점에 프로토타입 빈을 생성하고, 필요한 의존관계를 주입한다.



3. 스프링 컨테이너는 생성한 프로토타입 빈을 클라이언트에 반환한다.
4. 스프링 컨테이너에 같은 요청이 오면 항상 새로운 프로토타입 빈을 생성해서 반환한다.

- 스프링 컨테이너에 요청할 때 마다 빈이 새로 생성된다 --> **완전히 다른 스프링 빈이 생성된다.**
- 프로토타입 빈의 생성과 의존관계 주입 그리고 초기화까지만 관여 --> **종료 메서드가 호출되지 않는다.**

프로토타입 스코프 - 싱글톤 빈과 함께 사용시 문제점

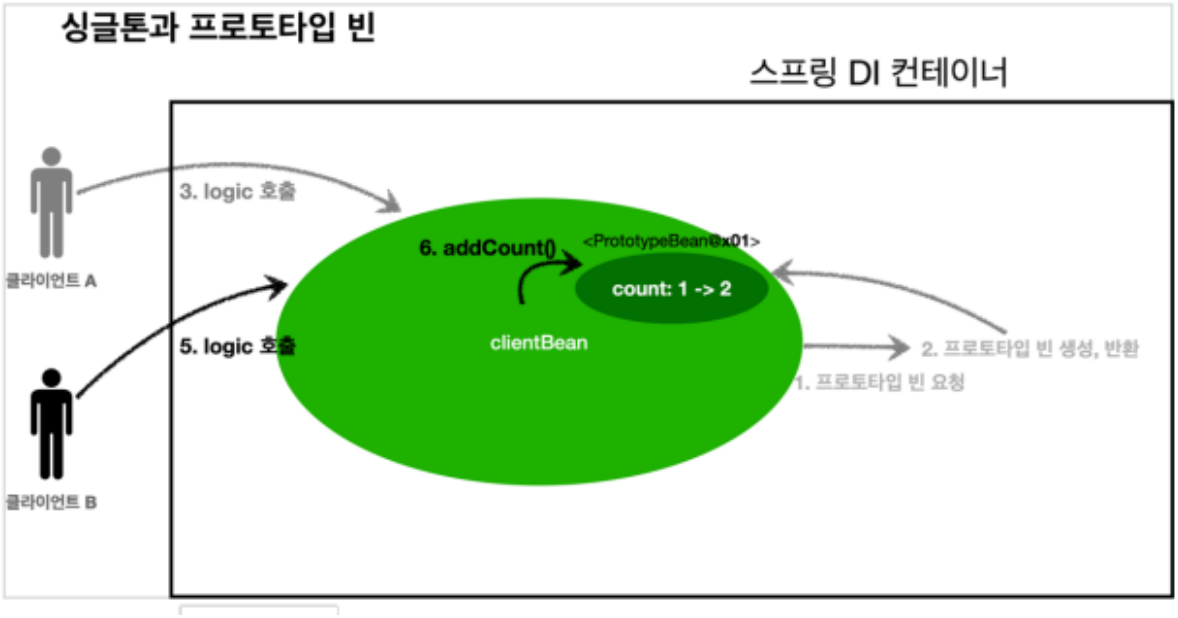
스프링은 일반적으로 싱글톤 빈을 사용하므로, 싱글톤 빈이 프로토타입 빈을 사용하게 된다.

그런데 싱글톤빈은 생성 시점에만 의존관계 주입을 받기 때문에, 프로토타입 빈이 새로 생성되기는 하지만,

싱글톤 빈과 함께 계속 유지되는 것이 문제다.(count를 2번했을 때 1, 1 이나오지 않고 2라는 결과가 출력)

- 주입 시점에 스프링 컨테이너에 요청해서 프로토타입 빈이 새로 생성이 된것, 사용 할 때마다 새로 생성되는게 아니다

프로토타입 빈을 주입 시점에만 새로 생성하는게 아니라 사용할때 마다 새로 생성해서 사용하는 것을 원할것이다.



프로토타입 스코프 - 싱글톤 빈과 함께 사용시 Provider로 문제 해결

간단한 방법: 싱글톤 빈이 프로토타입을 사용할 때 마다 스프링 컨테이너에 새로 요청하는 것

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

- ac.getBean()를 통해서 항상 새로운 프로토타입 빈이 생성된다.
- 직접 필요한 의존관계를 찾는 것을 Dependency Lookup(DL) 의존관계 조회(탐색) 이라고 한다.

ObjectFactory, ObjectProvider

- 지정한 빈을 컨테이너에서 대신 찾아주는 **DL 서비스**를 제공하는 것이 바로 **ObjectProvider** 이다.

```
@Autowired
private ObjectProvider<PrototypeBean> prototypeBeanProvider;
public int logic() {
    PrototypeBean prototypeBean = prototypeBeanProvider.getObject();
    prototypeBean.addCount();
    int count = prototypeBean.getCount();
    return count;
}
```

- prototypeBeanProvider.getObject()를 통해서 항상 새로운 프로토타입 빈이 생성된다.

JSR-330 자바표준 Provider

```
//implementation 'javax.inject:javax.inject:1' gradle 추가 필수
@Autowired
private Provider<PrototypeBean> provider;
public int logic() {
    PrototypeBean prototypeBean = provider.get();
    prototypeBean.addCount();
    int count = prototypeBean.getCount();
    return count;
}
```

- **provider 의 get()** 을 호출하면 내부에서는 스프링 컨테이너를 통해 해당 빈을 찾아서 반환한다. (DL)

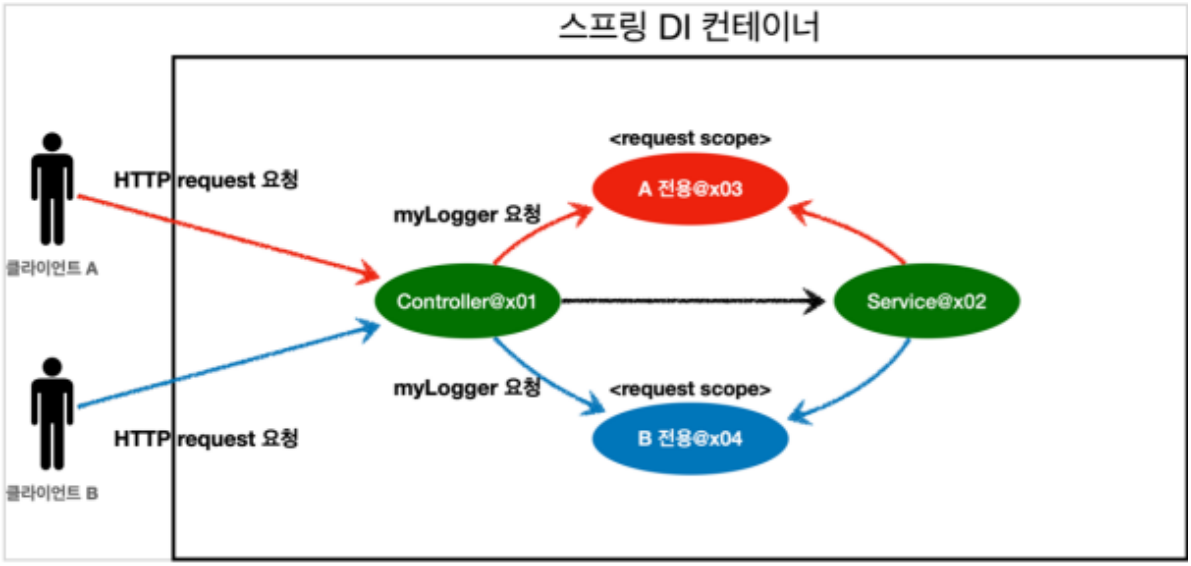
웹 스코프

- 웹 환경에서만 동작한다.
- 스프링이 해당 스코프의 종료시점까지 관리한다. --> 종료 메서드 호출

웹스코프 종류

- **request**: HTTP 요청 하나가 들어오고 나갈 때 까지 유지되는 스코프, 각각의 HTTP 요청마다 별도의 빈인스턴스가 생성되고, 관리된다.
- **session**: HTTP Session과 동일한 생명주기를 가지는 스코프
- **application**: 서블릿 컨텍스트(ServletContext)와 동일한 생명주기를 가지는 스코프
- **websocket**: 웹 소켓과 동일한 생명주기를 가지는 스코프

HTTP request 요청 당 각각 할당되는 request 스코프



- **동시에 여러 HTTP 요청**이 오면 정확히 어떤 요청이 남긴 로그인지 구분하기 어렵다. --> **request 스코프 사용**
- **@Scope(value = "request")** 를 사용해서 request 스코프로 지정
- HTTP요청 당 하나씩 생성되고 HTTP 요청이 끝나느 시점에 소멸된다.

스코프와 Provider

- 스프링 애플리케이션을 실행하는 시점에 싱글톤 빈은 생성해서 주입이 가능하지만,
- request 스코프 빈은 아직 생성되지 않는다. --> **실제 고객의 요청이 와야 생성할 수 있다.**

```
@Service
@RequiredArgsConstructor
public class LogDemoService {

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

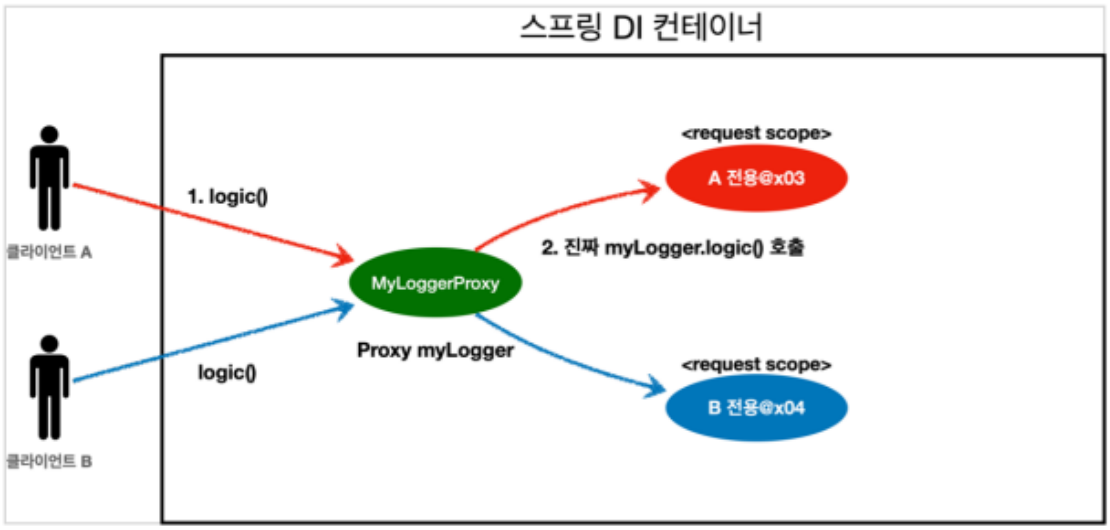
- **ObjectProvider** 덕분에 **ObjectProvider.getObject()** 를 호출하는 시점까지 r**equest scope 빈의 생성을 지연**
- ObjectProvider.getObject() 를 호출하시는 시점에는 HTTP 요청이 진행중이므로 request scope 빈의 생성이 정상처리

스코프와 프록시

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

- proxyMode = ScopedProxyMode.TAGRET_CLASS(INTERFACES) 를 사용한다.
- 가짜 프록시 클래스를 만들어두고 HTTP request와 상관없이 가짜 프록시 클래스를 다른 빈에 미리 주입해 둘 수있다.
- 스프링 컨테이너는 CGLIB라는 바이트코드를 조작하는 라이브러리를 사용해서, 가짜 프록시 객체를 생성한다.

```
myLogger = class hello.core.common.MyLogger$$EnhancerBySpringCGLIB$$b68b726d
```



- CGLIB라는 라이브러리로 내 클래스를 상속 받은 가짜 프록시 객체를 만들어서 주입한다.
- 가짜 프록시 객체는 실제 요청이 오면 그때 내부에서 실제 빈을 요청하는 위임 로직이 들어있다.
- 가짜 프록시 객체는 실제 request scope와는 관계가 없다. 내부에 단순한 위임 로지만 있고 싱글톤처럼 동작한다.