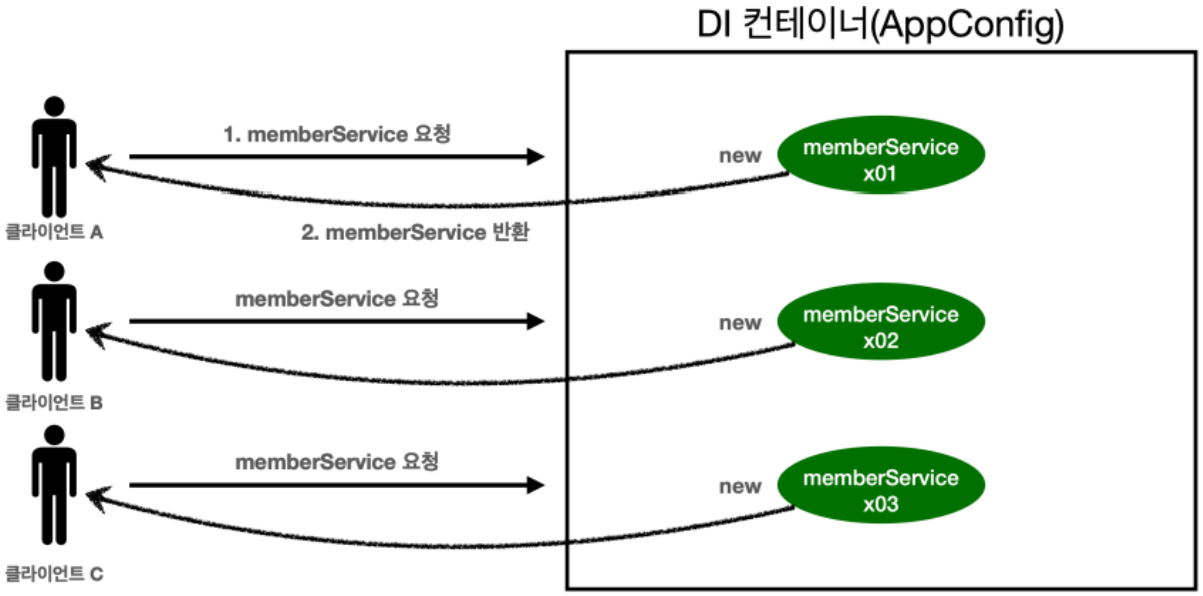
Section5. 싱글톤 컨테이너

싱글톤 패턴: 객체 인스턴스가 JVM 안에 1개만 존재해야 한다.

웹 애플리케이션과 싱글톤



- 위와 같은 방식은 고객의 요청이 올때마다 새로운 `MemberService` 객체가 만들어진다.

> 💡 스프링 없는 순수한 DI 컨테이너 테스트 코드 작성

- 과거의 순수한 DI 컨테이너인 AppConfig 는 요청을 할 때마다 객체를 새로 생성
  - 메모리 낭비가 심하다.
- 해결방안은 해당 객체가 딱 1개만 생성, 공유하도록 설계 → **싱글톤 패턴**

싱글톤 패턴

- 클래스의 인스턴스가 딱 1개만 생성되는 것을 보장하는 디자인 패턴이다.

    - `private` 생성자를 사용해서 외부에서 임의로 `new` 키워드를 사용하지 못하도록 막는다.
- 스프링을 사용하면 스프링이 알아서 스프링 컨테이너의 객체들에 싱글톤을 적용시킨다.

싱글톤 패턴 문제점

- 싱글톤 패턴을 구현하는 코드 자체가 많이 들어간다.
- 의존관계상 클라이언트가 구체 클래스에 의존 → **DIP 위반**
- 클라이언트가 구체 클래스에 의존해서 **OCP 원칙을 위반**할 가능성이 높다.
- 테스트하기 어렵다.
- 내부 속성을 변경하거나 초기화 하기 어렵다.
- `private` 생성자로 자식 클래스를 만들기 어렵다.
    - 유연성이 굉장히 떨어진다.
- 안티패턴으로 불리기도 한다.

싱글톤 컨테이너

지금까지 우리가 학습한 스프링 빈이 바로 싱글톤으로 관리되는 빈이다.

싱글톤 컨테이너

- 스프링 컨테이너는 singleton 패턴을 적용하지 않아도, 객체 인스턴스를 싱글톤으로 관리
- 스프링 컨테이너는 싱글톤 컨테이너 역할을 한다.
    - 이렇게 싱글톤 객체를 생성하고 관리하는 기능을 **싱글톤 레지스트리**라 한다.
- 기존 싱글톤 패턴의 모든 단점을 해결하면서 객체를 싱글톤으로 유지
    - 지저분한 코드 X
    - DIP, OCP, 테스트, private 생성자로 부터 자유롭게 싱글톤 사용

싱글톤 컨테이너 적용 후

- 이미 만들어진 객체를 공유해서 효율적으로 재사용

💡 스프링 의 기본 빈 등록 방식은 싱글톤이지만, 요청할 때 마다 새로운 객체를 생성해서 반환하는 기능도 제공

싱글톤 방식의 주의점

- 여러 클라이언트가 하나의 같은 객체 인스턴스를 공유하기 때문에 싱글톤 객체는 상태를 유지(stateful)하게 설계하면 안된다.

  → **무상태(stateless)**로 설계해야 한다.

  - 특정 클라이언트에 의존적인 필드가 있으면 안된다.
  - 특정 클라이언트가 값을 변경할 수 있는 필드가 있으면 안된다.
  - 가급적 읽기만 가능해야 한다.
  - 필드 대신에 공유되지 않는 지역변수, 파라미터, ThreadLocal 등을 사용해야 한다.
- 스프링 빈의 필드에 공유 값을 설정하면 정말 큰 장애가 발생할 수 있다.

💡 실습 코드 작성

- `StatefulService` 의 `price` 필드는 공유되는 필드
  - 사용자 A의 주문금액은 10000원이 되어야 하는데, 20000원이라는 결과가 나오게 된다.
- 스프링 빈은 항상 **무상태(stateless)**로 설계하자!

@Configuration과 싱글톤

`AppConfig` 코드를 보면`memberService` 와 `orderService` 두 개의 서비스에서 `new MemoryMemberRepository` 를 호출 하고 있다.

이러면 싱글톤 패턴이 깨진 것일까?

💡 테스트 코드 작성

```
memberService -> memberRepository = hello.core.member.MemoryMemberRepository@6691490c
orderService -> memberRepository = hello.core.member.MemoryMemberRepository@6691490c
memberRepository = hello.core.member.MemoryMemberRepository@6691490c
```

- 모두 같은 자원을 공유하고 있다.

```
call AppConfig.memberService
call AppConfig.memberRepository
call AppConfig.orderService
```

- `AppConfig` 의 메소드 호출 로그를 찍어보면 `memberRepository` 는 한번만 호출 된 것을 볼 수 있다.

> 💡 AppConfig 에서 생성자 주입을 하는 과정에 static을 붙이는 경우 참조값이 다른 것을 확인 할 수 있다.
>
> 이유는 스프링 컨테이너가
> `@Bean` 메서드를 오버라이딩하여 싱글톤을 보장하는 프록시 메커니즘을 사용하는 반면, `static` 메서드는 오버라이딩이 불가능하기 때문이다.
>
> https://www.inflearn.com/questions/1085291/static관련-질문드립니다

@Configuration과 바이트코드 조작의 마법

- 자바 코드를 보면 분명히 `MemberRepositroy` 는 3번 호출되어야 한다.
  - `@Configuration` 을 적용한 `AppConfig` 에 의해 1번만 호출 되었다.
- `AppConfig` 또한 스프링 빈이 된다.
  - `AppConfig` 의 클래스 정보를 출력해보면 다음과 같다.

```
// 예상과는 다르게 SpringCGLIB 이 붙어있는 것을 확인할 수 있다.
// CGLIB라는 바이트코드 조작 라이브러리를 사용해서
// AppConfig 클래스를 상속받은 임의의 다른 클래스를 만들고,
// 그 다른 클래스를 스프링 빈으로 등록한 것이다.
```

```
// 이 임의의 다른 클래스가 바로 싱글톤이 보장되도록 해준다.
bean = class hello.core.AppConfig$$SpringCGLIB$$0
```

- `AppConfig@CGLIB` 예상 코드

```
@Bean
public MemberRepository memberRepository() {

 if (memoryMemberRepository가 이미 스프링 컨테이너에 등록되어 있으
 return 스프링 컨테이너에서 찾아서 반환;
 } else { //스프링 컨테이너에 없으면
  기존 로직을 호출해서 MemoryMemberRepository를 생성하고 스프링 컨
  return 반환
 }
}
```

💡 `AppConfig@CGLIB`는 `AppConfig`자식 타입이므로, `AppConfig`타입으로 조회 가능하다.

정리

- `@Bean`만 사용해도 스프링 빈으로 등록되지만, 싱글톤을 보장하지 않는다.
  - `memberRepository()` 처럼 의존관계 주입이 필요해서 메서드를 직접 호출할 때
- 스프링 설정 정보는 항상 `@Configuration` 을 사용하자.