싱글톤 컨테이너 원리 이해하기

미션	[백엔드] 스프링 핵심 원리 - 기본
상태	완료
설명	싱글톤 패턴을 적용하여 동일한 인스턴스가 여러번 호출되어도 하나의 인스턴스로 동작하는 것을 실습해 봅니다. 실습 후 싱글톤의 장단점을 간단히 정리합니다. 결과물로 싱글톤 동작 결과 스크린샷 및 장단점 내용을 정리하고 PDF 문서로 만들어 제출합니다.

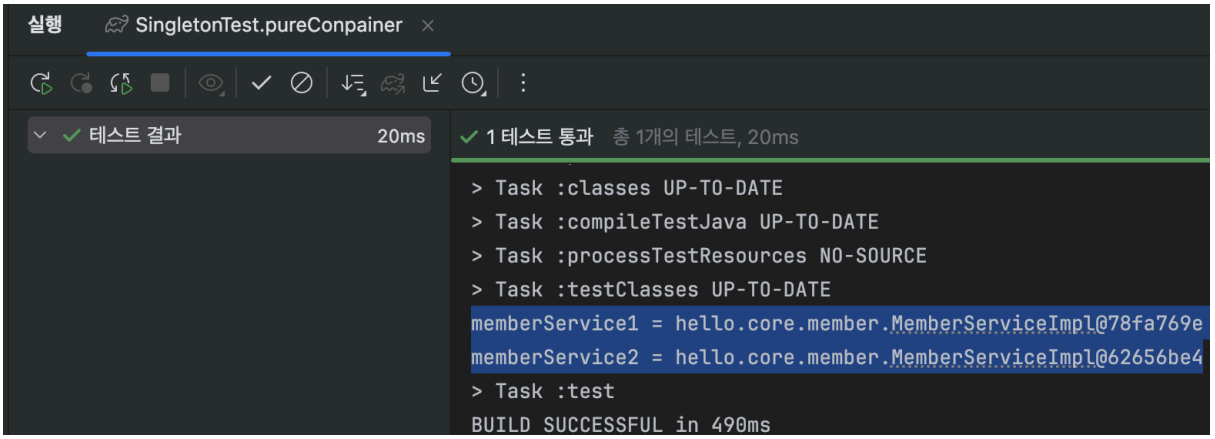
스프링 없는 순수한 DI 컨테이너

- 순수한 DI 컨테이너 `AppConfig`는 요청 마다 새로운 객체를 생성한다.
  - 100번의 요청이 발생하면 100개의 객체 생성 → 메모리 낭비
- 해당 문제를 해결하기 위해선 객체가 1개만 생성되게 하고, 이를 공유하도록 설계하면 된다. → **싱글톤 패턴**

SingletonTest.java

```
package hello.core.singleton;

import hello.core.AppConfig;
import hello.core.member.MemberService;
import org.assertj.core.api.Assertions;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

public class SingletonTest {

    @Test
    @DisplayName("스프링 없는 순수한 DI 컨테이너")
```

```
    void pureConpainer() {
        AppConfig ac = new AppConfig();

        // 호출 할 때 마다 객체 생성
        MemberService memberService1 = ac.memberService();
        MemberService memberService2 = ac.memberService();

        // 참조 값이 다른 것을 확인
        System.out.println("memberService1 = " + memberService1);
        System.out.println("memberService2 = " + memberService2);

        Assertions.assertThat(memberService1).isNotSameAs(memberService
2);
    }
}
```



다음의 테스트를 통해 AppConfig로 생성된 객체는 각기 다른 값을 참조한 다는 것을 알게 되었다.

싱글톤 패턴

- 클래스의 인스턴스가 딱 하나만 생성되는 것을 보장하는 디자인 패턴
- 따라서 객체 인스턴스를 2개 이상 생성하지 못하도록 막아야 한다.

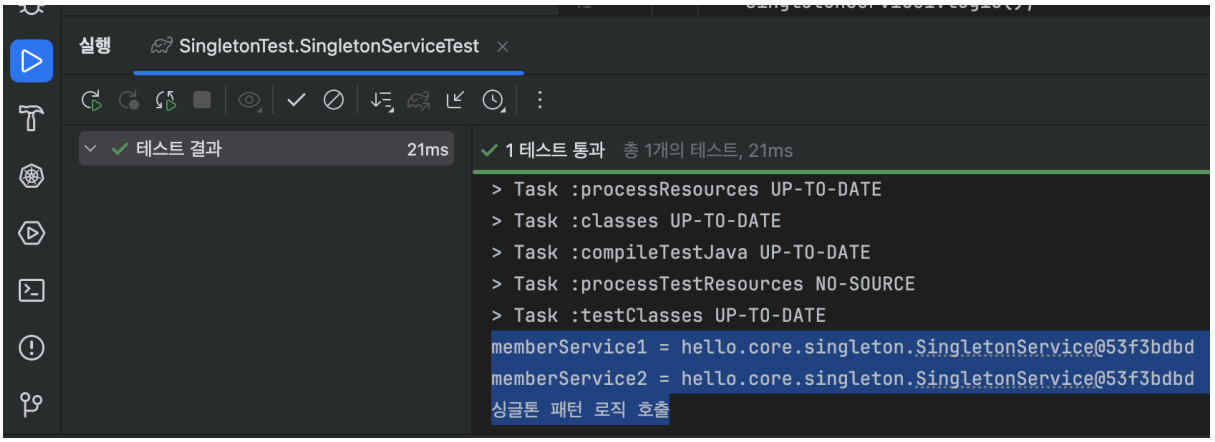
SingletonService.java

```
package hello.core.singleton;

public class SingletonService {

    // 1. static 영역에 객체를 하나만 생성한다.
    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 private로 선언하여 외부에서 new 키워드를 사용하지 못하게 막는다.
    private SingletonService() {

    }

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

- static 영역에 미리 인스턴스를 생성하고, 해당 인스턴스를 호출하기 위해선 `getInstance()`를 통해서만 호출할 수 있다.
- private를 이용하여 혹시라도 외부에서 new 키워드로 객체 인스턴스가 생성되는 것을 막는다.
    - 이 외에도 싱글톤 패턴을 구현하는 방식은 다양하다.

```
public class SingletonTest {

        ...

    @Test
```

```
    @DisplayName("싱글톤 패턴을 적용한 객체 사용")
    public void SingletonServiceTest() {
        // 호출 할 때 마다 같은 객체 생성
        SingletonService singletonService1 = SingletonService.getInstance();
        SingletonService singletonService2 = SingletonService.getInstance();

        // 참조 값이 같은지 확인
        System.out.println("memberService1 = " + singletonService1);
        System.out.println("memberService2 = " + singletonService2);

        Assertions.assertThat(singletonService1).isSameAs(singletonService
2);

        singletonService1.logic();
    }
}
```



싱글톤 패턴을 적용한 객체의 참조값은 같은 것을 알 수 있다.

싱글톤 패턴의 문제점

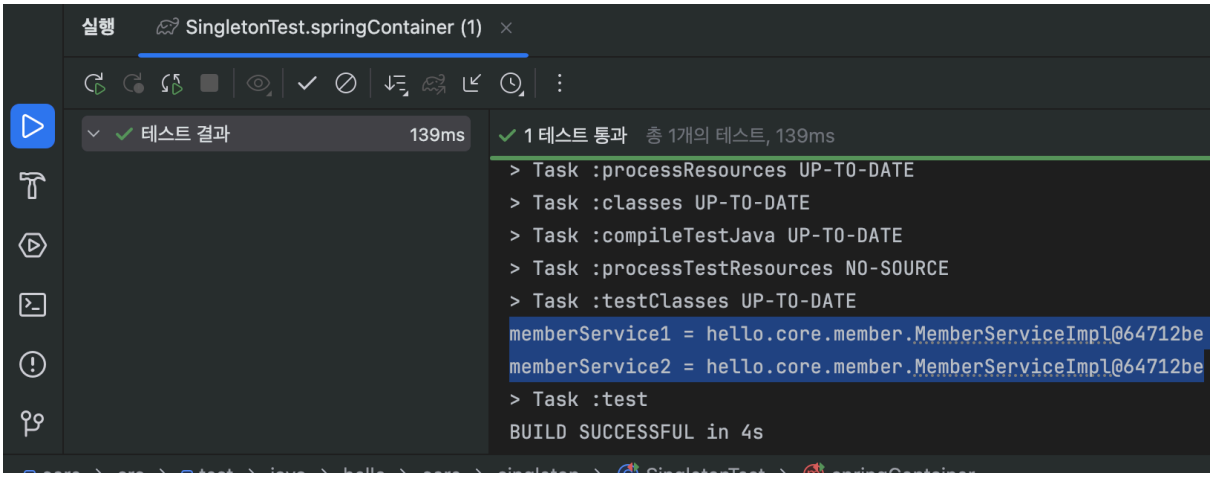
> 싱글톤 패턴을 적용하면 고객의 요청이 올 때 마다 객체를 생성하는 것이 아니라 이미 만들어진 객체를 공유해서 효율적으로 사용할 수 있지만, 다음과 같은 수 많은 문제점들을 가지고 있다.

- 싱글톤 패턴 구현 자체가 길다
- 의존관계상 클라이언트가 구체 클래스에 의존한다. (DIP 위반)
- 클라이언트가 구체 클래스에 의존하기 때문에 OCP를 위반할 가능성이 높다.
- 테스트 하기 어렵다.
- 내부 속성을 변경하거나 초기화 하기 어렵고, 자식 클래스를 만들기 어렵다. → 유연성이 떨어진다.

싱글톤 컨테이너

> 스프링 컨테이너는 싱글톤 패턴을 적용하지 않아도 객체 인스턴스를 싱글톤으로 관리한다.

```
public class SingletonTest {

    ...

    @Test
    @DisplayName("스프링 컨테이너와 싱글톤")
    void springContainer() {
        ApplicationContext ac = new AnnotationConfigApplicationContext(App
Config.class);

        MemberService memberService1 = ac.getBean(MemberService.clas
s);
        MemberService memberService2 = ac.getBean(MemberService.clas
s);

        System.out.println("memberService1 = " + memberService1);
        System.out.println("memberService2 = " + memberService2);

        Assertions.assertThat(memberService1).isSameAs(memberService2);
    }
}
```



스프링 컨테이너를 이용하여 싱글톤 방식을 구현할 수 있다!

싱글톤 방식 주의할 점

- 객체를 하나만 생성하여 공유하는 방식인 싱글톤은 여러 클라이언트가 하나의 객체를 공유하기 때문에 싱글톤 객체는 상태를 유지하도록 설계하면 안된다.
- 무상태로 설계해야 한다.
  - 특정 클라이언트에 의존적인 필드가 있으면 안된다.
  - 특정 클라이언트가 값을 변경할 수 있는 필드가 있으면 안된다.
  - 가급적 읽기만 가능해야 한다.

싱글톤 장단점

장점

1. 인스턴스 공유
   a. 객체를 하나만 생성하므로 메모리 관리에 용이하다.
   b. 여러 클래스에서 같은 객체를 참조한다.
2. 글로벌 접근
   a. 어디서든 동일한 인스턴스에 접근 가능

3. 상태관리 용이
    a. 애플리케이션 전역에서 상태를 일관되게 관리할 수 있다.

단점

1. 테스트가 어렵다.
2. 유연성이 부족하다.
3. DIP, OCP를 위반한다.