제어 역전 컨테이너와 의존성 주입

인터페이스 다시 보기

- 의존성
  - 클래스의 변경이 다른 클래스에 미치는 영향
  - 한 클래스가 다른 클래스의 메서드를 사용할 때 의존성 발생

- 클래스 간의 직접적인 의존성은 유지 보수 측면에서 문제 유발

- 인터페이스는 클래스 간의 의존성을 제거할 수 있는 문법 자원
  - 클래스 간의 직접 호출을 사용하지 않고 인터페이스를 통해 연결

- 인터페이스를 사용하더라도 객체 생성 구문에서 발생하는 의존성은 남음
  - 인터페이스는 참조 타입으로만 사용할 수 있고 new 연산자를 통해 인스턴스를 만들 수 없는 타입 → 결국 코드에 클래스 정보가 남아 의존성이 완전하게 제거되지 않음
  - 인스턴스 생성 로직을 코드로부터 분리해서 관리할 수 있는 방법 필요

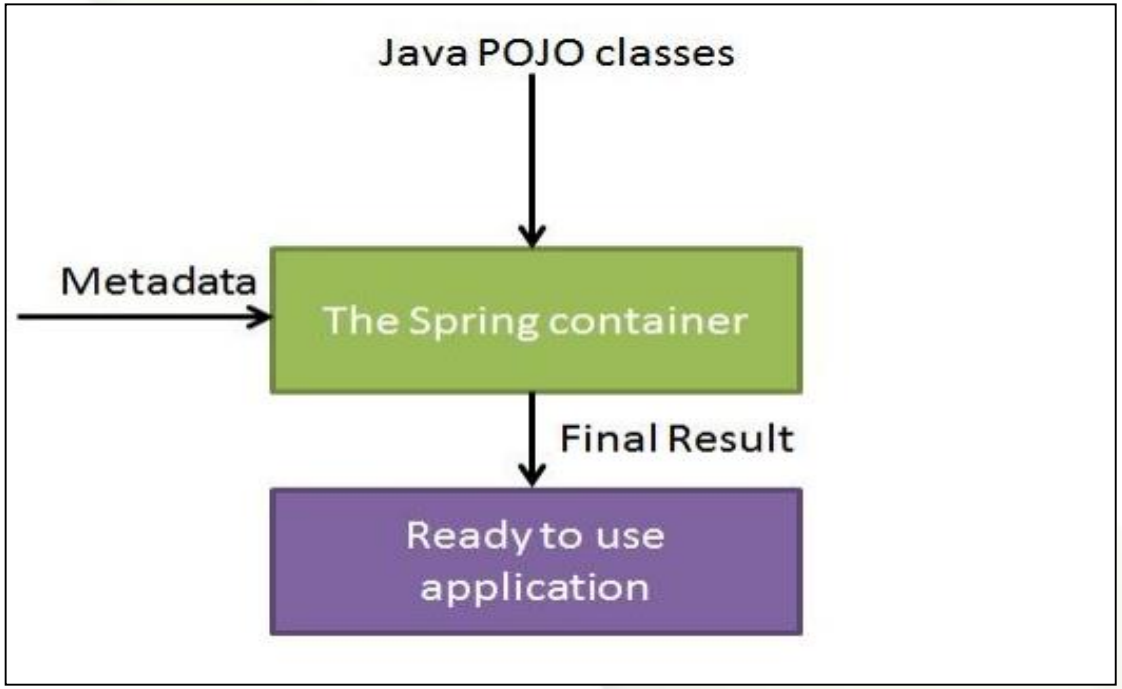
객체 팩토리 구현

리플렉션, 설정 파일 사용

➡ 객체 생성 및 관리를 구현 코드로부터 분리

Spring IoC Container

- 스프링 애플리케이션에서는 객체의 생성, 의존성 관리, 사용, 제거 등의 작업을 코드 대신 독립된 컨테이너가 담당



- 구성요소
  - 애플리케이션 컨텍스트
  - 관리 대상 POJO 클래스 집합 → 스프링 빈
  - 설정 메타 정보

IoC 컨테이너 종류

- Spring BeanFactory Container
  - 빈팩토리는 Bean 생성 및 DI 등의 Bean 관리에 집중하는 컨테이너
  - org.springframework.beans.BeanFactory 인터페이스 구현

- Spring ApplicationContext Container
  - 빈팩토리 기능에 다양한 엔터프라이즈 애플리케이션 개발 기능 추가 제공
  - org.springframework.context.ApplicationContext 인터페이스 구현
  - 스프링의 IoC 컨테이너는 일반적으로 애플리케이션 컨텍스트를 의미

BeanFactory 사용

```java
public class MessageService {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return String.format("Your
    }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        https://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/context
                        https://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="messageService" class="com.springexample.ioc.MessageService">

        <property name="message" value="Hello, Spring IoC !!!!!" />

    </bean>

</beans>
```

```java
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions("app-context.xml");

MessageService messageService = factory.getBean("messageService", MessageService.class);

String message = messageService.getMessage();

System.out.println(message);
```

ApplicationContext 사용

```java
public class MessageService {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return String.format("Your
    }

}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        https://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/context
                        https://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="messageService" class="com.springexample.ioc.MessageService">

        <property name="message" value="Hello, Spring IoC !!!!!" />

    </bean>

</beans>
```

```java
GenericXmlApplicationContext context = new GenericXmlApplicationContext("app-context.xml");

MessageService messageService2 = context.getBean("messageService", MessageService.class);

String message2 = messageService2.getMessage();

System.out.println(message2);

context.close();
```

빈 등록 방법

- 설정 XML 파일의 <bean> 태그

```
<bean id="beanName" class="패키지경로.클래스이름">
    생성자 주입 정보 또는 세터 주입 정보
</bean>
```

- 자바 코드에 의한 빈 등록 (@Configuration, @Bean)

```
@Configuration public class AnnotatedHelloConfig {

    @Bean public AnnotatedHello annotatedHello() {
        return new AnnotatedHelloConfig();
    }
}
```

- 자동인식을 이용한 빈 등록

```
@Component("bean-name")
public class AnnotatedHello { … }
```

```
<context:component-scan base-package="pacakge-name" />
```

XML 설정 방식

```
@ComponentScan(base-packages="package-name")
```

코드 설정 방식

스프링 빈 주요 속성 목록

속성	설명
class	생성, 관리 대상 Bean Class (필수)
id	생성, 관리 대상 Bean 이름
scope	생성된 객체의 유지 범위
constructor-arg	생성자 전달인자 (의존성 주입 도구)
property	setter 메서드 (의존성 주입 도구)
autowiring mode	의존성 주입 자동화 설정 (명시적 설정 없이 의존성 주입)
lazy-initialization mode	객체의 생성 시점 설정 (프로그램 시작 vs 첫 번째 객체 요청)
initialization method	Bean 객체 생성 후 호출될 초기화 메서드
destruction method	컨테이너가 소멸될 때 호출될 메서드

의존성 주입 (DI, Dependency Injection)

- IoC 컨테이너가 객체 간의 의존성 관리를 위해 사용하는 구현 기법
  - 객체가 필요한 곳에 스프링 컨테이너가 객체를 자동으로 할당
  - 객체의 변경에 유연한 코드 구현 가능

- 의존성 주입 방법
  - 생성자 주입 → 생성자 메서드를 사용해서 객체 할당
  - 세터 주입 → setXxx(…) 메서드를 사용해서 객체 할당
  - 기본 데이터 타입 설정 → 객체 의존성 주입과 같은 방법

다중 설정 파일 사용

- 관리 대상 객체가 많아지면 하나의 설정 파일에 관리하는 것이 어려움
  - 영역별로 객체를 나누고 여러 개의 설정 파일에서 관리

- 구현 방법
  - IoC 컨테이너 (XxxApplicationContext 객체)를 만들 때 여러 개의 설정 파일을 제공 → 각 설정 파일의 객체는 @Autowired 어노테이션으로 서로 의존성 주입 가능

  - Import 기능을 사용해서 다른 설정 파일의 내용을 포함

getBean() 메서드 사용

- IoC 컨테이너로부터 객체를 가져올 때 사용하는 메서드

- getBean("bean-id", bean-class) 형식
  - IoC 컨테이너가 관리하는 객체 중에서 id가 일치하는 객체 반환
  - 해당 bean-id로 관리되는 객체가 없을 경우 오류 발생
  - 반환되는 객체의 타입과 호환되지 않는 bean-class 를 지정한 경우 오류 발생

- getBean(bean-class) 형식
  - IoC 컨테이너가 관리하는 객체 중에서 타입이 bean-class와 호환되는 객체 반환
  - 컨테이너에 해당 타입과 호환되는 객체가 없을 경우 오류 발생
  - 컨테이너에 해당 타입과 호환되는 객체가 두 개 이상이 있을 경우 오류 발생

의존 객체 자동 주입

- IoC 컨테이너가 관리하는 객체 사이의 의존성을 코드를 통해 직접 주입하지 않고 암시적으로 주입하는 기법

- @Autowired, @Resource, @Inject 등의 어노테이션을 통해 암시적 의존성 주입
  - 필드(변수), setter 메서드, 생성자 등에 적용

  - 호환 가능한 bean이 여러 개인 경우 오류 발생
    - » @Qualifier 지정 또는 구체적 타입 지정 등을 통해 오류 해결

  - 일치하는 bean이 없는 경우 오류 발생
    - » required 속성을 false 로 지정
    - » Nullable 전달인자 또는 Optional 전달인자 사용

의존 객체 자동 주입

- 일치하는 bean이 없는 경우 오류 발생 (계속)
  - @Autowired에 required=false를 지정한 경우 대상 bean이 없으면 의존성 주입을 수행하지 않음
  - @Nullable, Optional을 지정한 경우 대상 bean이 없으면 null값을 의존성 주입

- 의존성 자동 주입과 명시적 의존성 주입이 동시에 적용된 경우
  - 의존성 자동 주입 설정이 적용됨
  - 특별한 이유가 없다면 일관된 방법을 사용하는 것이 권장됨

의존 객체 자종 주입 설정 Annotation

- Spring Annotation
  - @Autowired : 자동 의존성 주입 설정
  - @Qualifier : 의존성 주입 대상 빈을 명시적으로 지정

- JSR-330
  - @Inject (Spring Annotation의 @Autowired)
  - @Named (Spring Annotation의 @Qualifier)
  - @Value : 직접 값 주입

- JSR-250
  - @Resource (Spring Annotation의 @Autowired)
  - @PostConstruct : 스프링 빈 정의의 init-method 속성과 같은 기능
  - @PreDestroy : 스프링 빈 정의의 destroy-method 속성과 같은 기능

Annotation Based Bean Configuration

- 스프링은 Annotation을 이용한 빈 정의 및 의존성 주입 설정 지원

- 컨테이너에 Annotation 기반 빈 설정 활성화 필요
  - annotation-config 설정
    - » 이미 등록된 빈의 annotation (@Autowired, @Qualifier 등) 활성화

```
<context:annotation-config/>

<bean id="customerService" class="com.ensoa.order.service.CustomerServiceImpl" />
<bean id="customerRepository"  class="com.ensoa.order.repository.CustomerRepositoryImpl"/>
```

  - component-scan 설정
    - » 빈 자동 등록 및 관련 annotation 활성화

```
<context:component-scan base-package="spring" />
```

```
@Configuration
@ComponentScan(basePackages = {"spring"})
```

  - component-scan을 사용하는 경우 annotation-config는 불필요

빈 정의 Annotation

- @Component
  - 클래스가 스프링 bean임을 표시하는 범용 Annotation

```
@Component("customerService")
public class CustomerServiceImpl  implements CustomerService {
```

- @Service
  - 업무 로직을 구현하는 서비스 bean을 표시 @Component

```
//Component("customerService")
@Service("customerService")
public class CustomerServiceImpl  implements CustomerService {
```

- @Repository
  - 데이터 접근 논리 구현 bean을 표시하는 @Component

```
//@Component("customerRepository")
@Repository("customerRepository")
public class CustomerRepositoryImpl implements CustomerRepository {
```

- @Controller
  - Spring MVC 컨트롤러 클래스를 표시하는 @Component

스캔 대상에서 제외

- component-scan의 excludeFilters 속성으로 자동 등록 제외 대상 지정

- 필터 종류
  - 정규 표현식 사용 → FilterType.REGEX
  - AspectJ 패턴 사용 → FilterType.ASPECTJ

```
@Configuration
@ComponentScan(basePackages = {"spring", "spring2" },
    excludeFilters = {
            @Filter(type = FilterType.REGEX, pattern="spring\\..*Dao" ),
            @Filter(type = FilterType.ASPECTJ, pattern="spring.*Dao" ),
            @Filter(type = FilterType.ANNOTATION, classes = { ManualBean.class } ),
            @Filter(type = FilterType.ASSIGNABLE_TYPE, classes = { MemberDao.class } )

})
public class AppCtxWithExclude {
```

  » AspectJ 패턴을 사용하는 경우 aspectjweaver 의존 패키지 등록 필요

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>
```

스캔 대상에서 제외

- 필터 종류 (계속)
  - Annotation 사용 → FilterType.ANNOTATION

```
@Configuration
@ComponentScan(basePackages = {"spring", "spring2" },
    excludeFilters = {
            @Filter(type = FilterType.REGEX, pattern="spring\\..*Dao" ),
            @Filter(type = FilterType.ASPECTJ, pattern="spring.*Dao" ),
            @Filter(type = FilterType.ANNOTATION, classes = { ManualBean.class } ),
            @Filter(type = FilterType.ASSIGNABLE_TYPE, classes = { MemberDao.class } )
})
public class AppCtxWithExclude {
```

```
@ManualBean
@Component
public class MemberDao {
```

```
@Retention(RUNTIME)
@Target(TYPE)
public @interface ManualBean {

}
```

스캔 대상에서 제외

- 필터 종류 (계속)
  - 특정 타입 및 하위 타입 지정 → FilterType.ASSIGNABLE_TYPE

```
@Configuration
@ComponentScan(basePackages = {"spring", "spring2" },
    excludeFilters = {
            @Filter(type = FilterType.REGEX, pattern="spring\\..*Dao" ),
            @Filter(type = FilterType.ASPECTJ, pattern="spring.*Dao" ),
            @Filter(type = FilterType.ANNOTATION, classes = { ManualBean.class } ),
            @Filter(type = FilterType.ASSIGNABLE_TYPE, classes = { MemberDao.class } )

})
public class AppCtxWithExclude {
```

Component Scan 과정에서 발생한 Bean 충돌 처리

- @Component로 등록한 2개 이상의 Bean이 암시적으로 같은 이름을 사용하는 경우 오류 발생 → 명시적으로 bean 이름을 다르게 지정해서 충돌을 피할 수 있음

- 수동으로 등록한 Bean과 @Component로 등록한 Bean이 같은 이름을 사용하는 경우 수동으로 등록한 Bean이 사용됨

IoC Container Life Cycle

```
//컨테이너 초기화
AnnotationConfigApplicationContext ctx =
        new AnnotationConfigApplicationContext(AppContext.class);

//컨테이너 사용
Greeter g = ctx.getBean("greeter", Greeter.class);
String msg = g.greet("스프링");
System.out.println(msg);

//컨테이너 종료
ctx.close();
```

Spring Bean Life Cycle

- Spring Bean의 생성 → 사용 → 소멸 과정의 중요한 시점 마다 컨테이너가 적절한 메서드 호출.

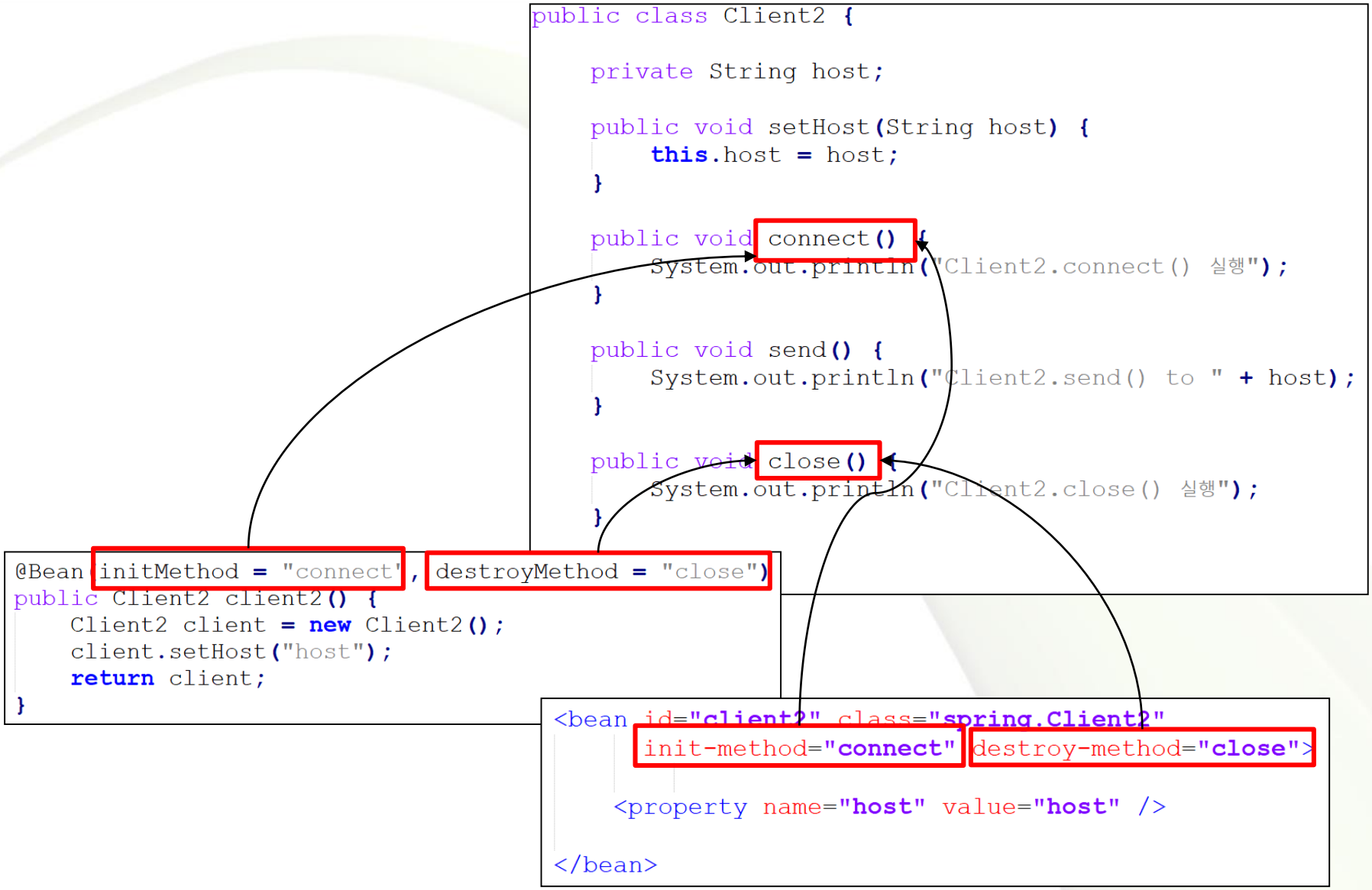
- 이러한 콜백으로 객체의 Life Cycle 관리

- 주로 초기화 및 종료 시점의 이벤트로 활용

- 구현 방법
  - 인터페이스 구현
    - » InitializingBean → 초기화, afterPropertiesSet
    - » DisposableBean → 종료 처리, destroy

  - 커스텀 메서드
    - » initMethod, destroyMethod 속성 사용

Spring Bean Life Cycle



Spring Bean Scope

- Spring bean에 지정된 scope에 따라 객체의 라이프사이클과 공유 범위가 결정됨.

- Scope 종류

Scope	설명
singleton	컨테이너 단위로 객체를 하나만 생성해서 모든 Bean들이 공유
prototype	객체의 요청이 있을 때 마다 새로운 객체 생성
request	웹 애플리케이션의 경우 요청 라이프사이클 범위
session	웹 애플리케이션의 경우 세션 라이프사이클 범위

- IoC 컨테이너는 prototype scope bean의 소멸은 관리하지 않음 → 소멸 처리는 직접 구현

Spring Bean Scope (Singleton)

```java
@Scope("singleton")
public Client2 client2() {
    Client2 client = new Client2();
    client.setHost("host");
    return client;
}
```

```xml
<bean id="client2" class="spring.Client2" scope="singleton">
    <property name="host" value="host" />
</bean>
```

```java
public static void main(String[] args) throws IOException {
    AbstractApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppCtxWithPrototype.class);

    Client2 client1 = ctx.getBean(Client2.class);
    Client2 client2 = ctx.getBean(Client2.class);
    System.out.println("client1 == client2 : " + (client1 == client2));

    ctx.close();
}
```

true

Spring Bean Scope (prototype)

```java
@Scope("prototype")
public Client2 client2() {
    Client2 client = new Client2();
    client.setHost("host");
    return client;
}
```

```xml
<bean id="client2" class="spring.Client2" scope="prototype">
    <property name="host" value="host" />
</bean>
```

```java
public static void main(String[] args) throws IOException {
    AbstractApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppCtxWithPrototype.class);

    Client2 client1 = ctx.getBean(Client2.class);
    Client2 client2 = ctx.getBean(Client2.class);
    System.out.println("client1 == client2 : " + (client1 == client2));

    ctx.close();
}
```

false