제어의 역전(Ioc)와 의존성 주입 (DI)

스프링 컨테이너가 필요에 따라 Bean들을 관리하거나 제어하는 행위.

제어의 역전은 프로그램의 제어 흐름을 직접 제어하는 것이 아니라 외부에서 관리하는 것을 의미한다. 기존 프로그램은 클라이언트 구현 객체가 스스로 필요한 서버 구현 객체를 생성하고, 연결하고 실행한다. 한마디로 구현 객체가 프로그램의 제어 흐름을 스스로 조종하는 것.

예로 할인 정책이 들어간 제품을 구매하는 상황이다.

실질적으로 OrderService의 구현 객체인 OrderService에서 주문이 발생함.

인자로 구매자의 나이, 제품의 이름, 제품의 가격을 입력하면 Order 객체를ㄹ 반환한다.

다만 OrderServiceImpl은 DiscountPolicy를 의존합니다. 그리고 이 할인 정책 종류에 따라

할인 금액이 정해집니다. DiscountPolicy와 관련된 객체들은 아래와 같다.

FixDiscountPolicy는 20살 미만인 이용자에게 1000원 할인, RateDiscountPolicy는 20살 미만인 이용자에게 10%할인을 적용한다고보자. 현재는 고정 할인정책을 수용.

Public class Main {

Public static void main(String[] args) {

final OrderService orderService = new OrderServiceimp();

final Order order = orderService.createOrder(10, “샤프”, 3000);

System.out.println(order.getDiscountPrice());

}

}

사용자 입장에서는 OrderService 객체를 만들고 주문을 진행하여 할인된 금액을 알아낸다.

**OrderServiceImpl은 DiscounPolicy와 FixDiscountPolicy를 모두 의존**하고 있습니다.

만약 할인 정책이 고정 할인 정책이 아니라 정률 할인 정책으로 바뀐다면 OrderService Impl의 코드를 수정합니다. 지금은 코드가 짧아서 단순히 ‘new FixDiscountPolicy()’ 부분을 ‘new RateDiscountPolicy()’로 고치기만 하면 된다. 하지만 만약에 해당 고정할인 정책을 의존하는 객체가 많고 로직이 많다면 그 부분을 싹다 고쳐야됨.

이를 해결하기 위한 가장 좋은 방법은 OrderServiceImpl 내에서 DiscountPolicy의 구현 객체를 정해주지 않는 것. 그렇다고 코드를 작성하면 NullPointerException이 발생.

따라서 외부에서 DiscountPolicy를 주입해주어야한다. 이때 의존성 주입(DI)가 사용되는데

현재 상황에서는 생성자를 통해 DiscountPolicy 초기화.

**public** **OrderServiceImpl**(DiscountPolicy discountPolicy) {

**this**.discountPolicy = discountPolicy;

}

추가

다만 이러면 다른 곳에서 컴파일 에러 발생.

사용자 입장에서 OrderService를 사용할 때 구현 객체를 정해줘야 한다.

이 부분도 매번 특정 객체를 만들어서 넘겨주어야 한다는 문제가 있다.

그래서 해당 객체를 주입해 주는 역할만 하는 설정 관련 객체를 새로 정의해보자.

AppConfig는 프로그램 전반적으로 사용될 객체를 미리 정의해 둔다.

그리고 사용자 입장에서는 무슨 할인 정책을 쓸지 선택할 필요가 없다.

단순히 AppConfig로부터 제공 받는 객체를 사용할 뿐이다.

만약 할인 정책이 바뀐다면, AppConfig에서 FixDiscountPolicy를

RateDiscountPolicy로 바꾸면 된다.

**public** **OrderServiceImpl**(DiscountPolicy discountPolicy) {

**this**.discountPolicy = discountPolicy;

}

**@Override**

**public** Order **createOrder**(**int** age, String itemName, **int** itemPrice) {

**int** discountPrice = discountPolicy.discount(age, itemPrice);

**return** **new** Order(itemName, itemPrice, discountPrice);

}

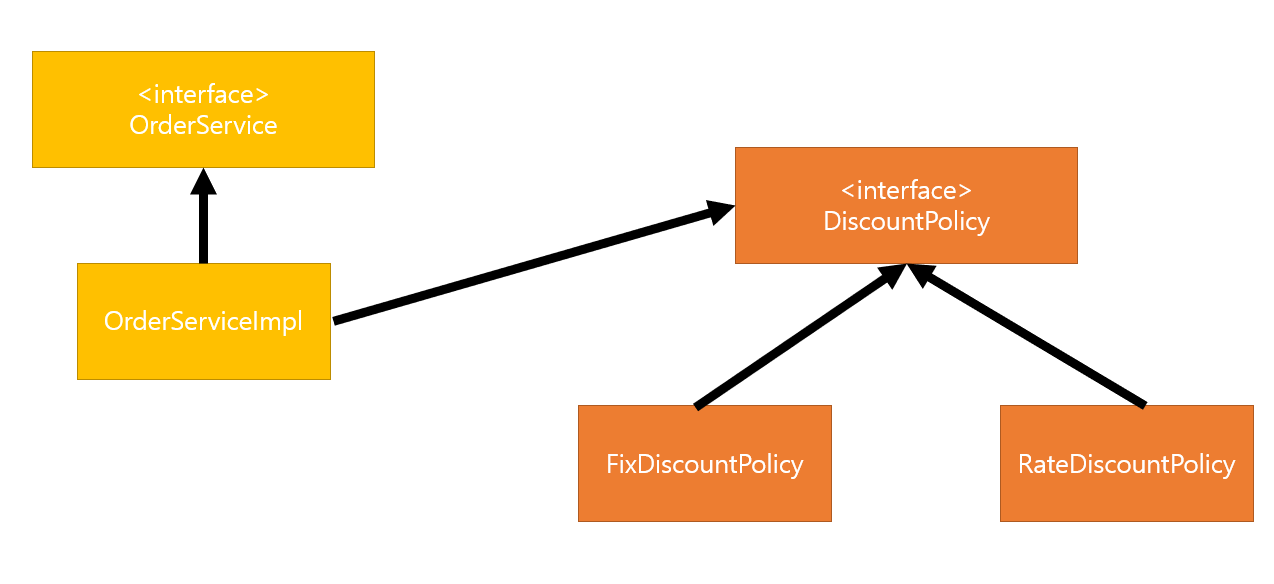
}

OrderServiceImpl은 할인 정책이 바뀌어도 상관이 없다. 애초부터 OrderServcieImpl은 주문만 하면 되는 것이지. 구체적으로 할인 정책을 알 필요가 없는 것이죠.

정리하면 구현 객체는 자신의 로직을 실행하는 역할만 담당하고 프로그램 제어 흐름은 AppConfig가 가져가게 되었습니다. 예를 들어서 OrderServiceImpl은 필요한 인터페이스들을 호출하지만 어떤 구현 객체들이 실행되는지는 모른다.또한, OrderServiceImpl 자체도 AppConfig가 생성하고 있으므로 Appconfig는 OrderServiceImpldl 이아닌 OrderService의 또다른 하위 구현 객체를 정의하여 사용자에게 넘겨준다. 이렇듯 프로그램의 제어흐름을 개발자가 직접하는 것이 아니라 외부에서 하는 것을 제어의 역전이라고 하며. 이 예시에서 외부는

AppConfig이다.

의존성 주입= 의존 관계 주입.

의존 관계는 정적인 클래스 의존 관계와 실행 시점에 결정되는 동적인 객체 의존 관계 둘로 분리해야한다. 위 예시에서 정적인 클래스 의존 관계

정적인 클래스 의존 관계는 클래스가 사용하는 import 코드만 보고 판단할 수 있다. 여기서 OrderServiceImpl은 DiscountPolicy에 의존한다. 하지만 OrderServiceImpl은 해당 정적 클래스 의존 관계만으로는

구체적으로 어떠한 구현 할인 객체가 주입 될지는 알 수가 없다.

그래서 우리는 동적인 객체 의존 관계를 파악해야한다.

동적인 객체 의존 관계는 애플리케이션 실행 시점에 생성된 객체 인스턴스의 참조가 연결된 의존관계를 의미한다. 그리고 애플리케이션 실행 시점에

외부에서 실제 구현 객체를 생성하고 클라이언트에 전달해서 클라이언트와

서버의 실제 의존관계가 연결되는 것을 의존 관계 주입이라한다.

위의 예시에서는 AppConfig에서 구현 할인 객체를 만들어서 OrderServiceImpl의 생성자로 넘겼습니다.

의존 관계 주입을 이용하면 클라이언트단의 코드를 변경하지 않고, 클라이언트가 호출하는 대상 타입의 인스턴스를 변경할 수있다. 즉, 정적인 클래스 의존 관계는 유지하되, 동적인 객체 의존 관계만 변경 된다.

우리가 할인 정책을 고정에서 정률로 바꾼다고 해서 OrderServiceImpl과 DiscountPolicy의 의존 관계가 깨지는 것도 아니기 때문.

지금 예제에서는 의존성 주입을 생성자 방식을 사용하였지만, setter문, 스프링의 경우 @Autowired 어노테이션을 사용하기도한다.

DI 컨테이너 AppConfig처럼 객체를 생성하고 관리하면서 의존 관계를

연결해 주는 것을 IOC컨테이너 혹은 DI컨테이너라고한다.

이제 이 DI가 언제 쓰이는지 계속해서 이야기할 것.

스프링 컨테이너란?

스프링 컨테이너는 자바 객체의 생명 주기를 관리하며, 생성된 자바 객체들에게 추가적인 기능을 제공하는 역할을 한다. 여기서 자바 객체를 스프링에서는

Bean이라고 부른다. 그리고 저번 시간에 배웠던 Ioc와 DI의 원리가 이 스프링 컨테이너에 적용된다.

개발자는 new 연산자, 인터페이스 호출, 팩토리 호출 방식으로 객체를 생성하고 소멸시킬 수 있는데, 스프링 컨테이너가 이 역할을 대신해 준다. 즉 제어 흐름을 외부에서 관리하는 것이다. 또한, 객체들 간의 의존관계를 스프링 컨테이너가 런타임 과정에서 알아서 만들어준다.

DI는 생성자 setter, @Autowired 를 통해 적용

스프링 컨테이너에는 BeanFactory와 ApplicationContext가 있다.

1. BeanFactory는 빈을 등록하고 생성하고 조회하고 돌려주는 등 빈을 관리하는 역할을 한다. getBean()메소드를 통해 빈을 인스턴스화 할 수 잇다.

먼저

**@Configuration**

**public** **class** **AppConfig** {

**@Bean**

**public** OrderService **orderService**() {

**return** **new** OrderServiceImpl(discountPolicy());

}

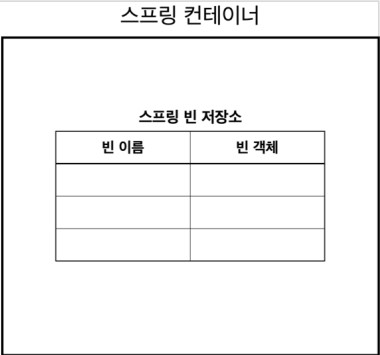
**@Bean**

**public** FixDiscountPolicy **discountPolicy**() {

**return** **new** FixDiscountPolicy();

}

}

해당 코드에서 @Bean이 붙은 메소드의 명을 스프링 빈의 이름으로 사용하여 빈 등록한다.

이런 식으로 스프링 컨테이너 안에 스프링 빈 저장소가 있고, 그 안에 빈이 들어있다.

그리고위와같이 BeanFactory AnnotationConfigApplicationContext로 정의하되, AppConfig를

구성 정보로 지정한다. 기존에는 개발자가 직접 AppConfig를 사용해서 필요한 객체를 직접 조회했지만, 이제부터는 스프링 컨테이너를 통해서

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** BeanFactory beanFactory = **new** AnnotationConfigApplicationContext(AppConfig.class);

**final** OrderService orderService = beanFactory.getBean("orderService", OrderService.class);

**final** Order order = orderService.createOrder(15, "샤프", 3000);

System.out.println(order.getDiscountPrice());

}

}

필요한 스프링 빈 객체를 찾을 수 있다.

getBean() 메소드에는 여러가지 인자가 들어갈 수 있는데 이번 예제에서는 빈의 이름과 타입을 지정하여 빈 객체를 구현해보자.

만약 Orderservice.class를 적어주지 않는다면 orderservice의 타입은 OrderService가 아니라 Object가 됨을 유의하자.

실행하면 로그가 쫙 뜬다. 중요한 것은

‘Creating shared instance of singleton bean’부분이다.

뭔가 긴 빈 이름은 미리 정의된 초기 빈들이고. appConfig, orderService, discountPolicy가 우리 등록한 빈이라는 사실을 알 수 있다. 또한, OrderService는 DiscountPolicy를 주입해주어야 하는데.

스프링 컨테이너가 알아서 주입을 해준다.

1. ApplicationContext

ApplicationContext도 BeanFactory처럼 빈을 관리할 수 있다. Main 코드에서 BeanFactory 를 ApplicationContext로만 바꾸고 실행해 보자.

**public** **class** **Main** {

**public** **static** **void** **main**(String[] args) {

**final** ApplicationContext beanFactory = **new** AnnotationConfigApplicationContext(AppConfig.class);

**final** OrderService orderService = beanFactory.getBean("orderService", OrderService.class);

**final** Order order = orderService.createOrder(15, "샤프", 3000);

System.out.println(order.getDiscountPrice());

}

}

동일한 결과를 얻는다.

BeanFactory vs ApplicationContext

위의 예제로만 보면 둘이 하는 역할이 같은데, 왜 두 개로 구분한 것인 것인지 이해가 안갈 수도 있다. 이것은 ApplicationContext가

BeanFactory의 상속을 받았기 때문

컴포넌트 스캔과 의존 관계 자동 주입.

@Configuration이 붙은 설정 파일을 이용하여 빈을 수동 주입하였다. 이렇게 개발자가 수동으로 관계를 정해줄 수도 있지만, 만약 등록할 빈이 많다면 일일이 모두 등록해야 해서 상당히 귀찮다. 그래서 스프링은 설정정보가 없어도 자동으로 스프링 빈을 등록하는 컴포넌트 스캔이라는 기능을 제공한다.

@Component

Public class OrderServiceImpl implements OrderService {

Private final DiscountPolicy discountPolicy;

Public OrderServiceImpl(DiscountPolicy discountPolicy) {

This.discountPolicy = discountPolicy;}

@Override

Public order createOrder

빈으로 등록할 클래스에 @Component를 붙여주고 설정파일에 @Configuration과 @ComponentScan을 붙여주면 끝이난다. @ComponentScan이 @Component가 붙은 모든 객체를 찾아서 빈으로 등록하기 때문.

사용자 입장에서는 동일하게 설정 파일을 이용하여 스프링컨테이너를 정의하고 비즈니스 로직을 작성하면 된다.