# DI와 IoC컨테이너

# Spring Framwork


- 스프링은 자바 엔터프라이즈급 애플리케이션 개발을 위한 프레임워크입니다. 객체지향 원칙을 지키면서 효율적이고 유연한 구조를 갖춘 애플리케이션을 개발할 수 있도록 다양한 기능을 제공합니다.

- 이번 포스트에서는 스프링의 3대 요소라 불리는 IoC/DI, PSA, AOP중에서 IoC/DI에 대해서 알아보겠습니다.

# IoC(제어의 역전)

- IoC은 Inversion of Control의 약자로 프로그램의 제어흐름의 구조가 바뀐다는 뜻입니다. 제어의 역전은 서블릿이나, AOP 구현, Mock 오브젝트 등 많은 데에서 사용되고 있습니다.

# IoC(제어의 역전)

- 서블릿을 생각해봅시다. 기존 자바 애플리케이션에서는 개발자가 main() 메서드에서 직접 객체를 생성하고 호출하며, 흐름을 제어했습니다. 하지만 서블릿에서는 사용자가 브라우저에서 요청을 보내면, 서블릿 컨테이너(Tomcat 등)가 요청을 받아 doGet()이나 doPost()를 자동으로 호출합니다. 즉 개발자 코드가 흐름을 제어하지 않고 컨테이너가 개발자 코드를 호출하는 구조로 흐름이 바뀐 겁니다.

- 프레임워크도 제어의 역전이 적용된 예입니다. 프레임워크는 애플리케이션 코드가 프레임워크에 의해 관리됩니다. 즉 프레임워크가 흐름을 주도하는 중에 개발자 애플리케이션 코드를 사용한다는 것입니다.

# 프레임워크와 라이브러리 차이

- 라이브러리: 작성한 코드가 제어흐름을 담당
- 프레임워크: 프레임워크가 흐름을 주도하면서 코드를 사용

# IoC 컨테이너

- 스프링은 애플리케이션개발을 위해 다양한 기능을 제공하지만, 핵심을 담당하는 것은 IoC컨테이너입니다.

- 컨테이너는 말 그대로 무언가를 담고 있는 상자입니다.
- IoC 컨테이너는 애플리케이션에서 사용할 객체를 담고 관리하는 객체 관리자입니다.

# IoC 컨테이너

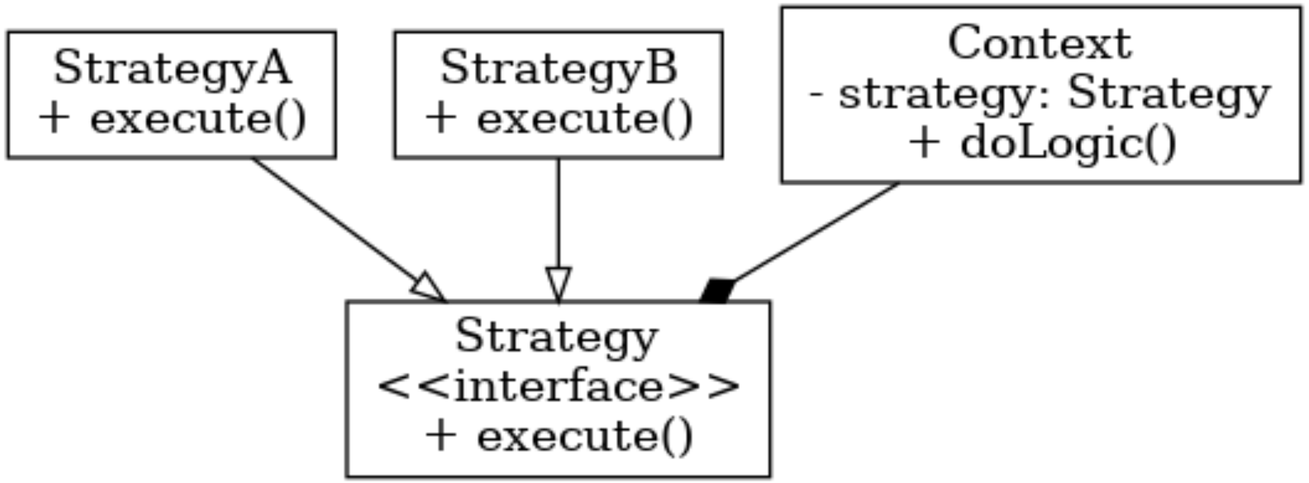
- 스프링에서 IoC 컨테이너는 Bean이라 불리는 객체를 담고 관리합니다.
- Bean은 스프링이 대신 생성하고 필요한 곳에 주입해주기 때문에, 제어의 역전 개념이 적용된 객체라고 할 수 있습니다. 빈은 밑에서 더 상세하게 설명 드리겠습니다.
- 스프링 IoC컨테이너는 BeanFactory와 ApplicationContext로 나눌 수 있습니다.

1. BeanFactory: 가장 기본적인 컨테이너. Bean을 생성하고, 필요할 때 제공함 (지연 로딩)
2. ApplicationContext: BeanFactory를 확장한 컨테이너. 국제화, 이벤트, AOP 등 추가 기능 포함 (즉시 로딩)

# DI

- DI는 Dependency Injection의 약자로 의존성 주입이라는 뜻을 가지고, 객체가 의존하는 다른 객체(의존성)를 직접 생성하지 않고, 외부에서 주입받는 것을 말합니다.

# DI


StrategyA
+ execute()
StrategyB
+ execute()
Context
- strategy: Strategy
+ doLogic()
Strategy
<<interface>>
+ execute()

# DI

```java
public interface Strategy {
    public void execute();
}


public class StrategyA implements Strategy {

    @Override
    public void execute() {
        System.out.println("A");
    }
}


public class StrategyB implements Strategy {

    @Override
    public void execute() {
        System.out.println("B");
    }
}
```

이제 Context 역할을 할 클래스를 만들고,

그 안에 Strategy를 DI 방식으로 주입해봅니다.

# DI

1. 생성자 주입: 가장 권장되는 방식입니다.

```
public class Context {

    private final Strategy strategy;

    // 생성자 주입
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void doLogic() {
        strategy.execute();
    }
}
```

# DI

## 2. Setter 주입: 의존성을 나중에 추가할 수 있어 유연합니다.

```
public class Context {

    private Strategy strategy;

    // Setter 주입
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void doLogic() {
        if (strategy == null) {
            throw new IllegalStateException("Strategy not set!");
        }
        strategy.execute();
    }
}
```

# DI

3. 필드 주입: 의존성을 나중에 추가할 수 있어 유연합니다.

```
public class Context {

    // 필드 주입
    public Strategy strategy;

    public void doLogic() {
        if (strategy == null) {
            throw new IllegalStateException("Strategy not set!");
        }
        strategy.execute();
    }
}
```

# DI

Context 클래스에서 어느 부분에서도 어떤 Strategy 클래스를 써야 할지 드러나지 않습니다.

이는 Context가 StrategyA인지 StrategyB인지에 대해 전혀 모른다는 것이며, 그 덕분에 코드가 유연하고 확장 가능해집니다.

즉, Context는 Strategy라는 추상(인터페이스)에만 의존하고, 구체적인 구현 클래스(StrategyA, StrategyB)는 외부에서 주입되기 때문에

전략을 바꾸더라도 Context 코드를 수정할 필요가 없습니다. **이처럼 외부에서 의존관계를 주입해주는 방법을 DI라고 부릅니다.**

- **스프링에서는 이 역할을 IoC 컨테이너가 담당합니다. 스프링은 의존성을 주입할 빈 객체를 만들고 스프링 IoC컨테이너에 넣어 사용할 수 있습니다.**

# Bean

- Bean은 스프링 컨테이너에 의해 관리되는 객체를 뜻합니다. 즉 컨테이너에 등록되어 있는 객체는 Bean 입니다.

# Bean

```
@Configuration
public class StrategyConfig {

    @Bean
    public Strategy strategy() {
        // 여기서 어떤 전략을 사용할지 결정
        return new StrategyA(); // 또는 new StrategyB();
    }

    @Bean
    public Context context(){
        return new Context(strategy());
    }
}
```

@Bean어노테이션을 사용하여 빈 객체를 컨텍스트에 저장하는 코드입니다.

참고로 @Bean으로 컨텍스트에 빈을 등록하는법 외에 @Component, @Component의 하위 어노테이션인 @Repository, @Service, @Controller, Xml로 등록하기 등 다양한 방법으로 빈을 등록할 수 있습니다.

빈을 등록하게 되면 **@Autowired로 의존성을 주입하여 빈을 가져와 쓸 수 있습니다.**

# Bean

```
@RequiredArgsConstructor
public class Context {
    private final Strategy strategy;

    public void doLogic() {
        strategy.execute();
    }
}
```

@RequiredArgsConstructor는 **final, @NonNull이 붙은 필드의 생성자를 자동으로 생성**해주는 어노테이션입니다.

또한 스프링 4.3 부터는 **생성자가 하나만 있으면 @Autowired 없이도 자동으로 의존성을 주입**해줍니다.

만약에 @RequiredArgsConstructor가 붙지 않으면

```
@Autowired
public Context(Strategy strategy){
    this.strategy = strategy;
}
```

를 통해 의존성을 주입할 수 있습니다.

# Bean

- 스프링에서는 **@RequiredArgsConstructor를 통한 생성자 주입 방식을 권장**하고 있습니다.

첫번 째 이유로 final 키워드를 사용하므로, **객체가 생성된 이후에는 필드를 바꿀 수 없습니다.** 이는 버그를 줄이고, 멀티스레드 환경에서도 더 안전합니다.

두번 째 이유로 **필드 주입시 발생하는 순환 참조를 예방**할 수 있습니다.

# Bean

```java
public class A {
    @Autowired private final B b;

    public A(B b) {
        this.b = b;
    }
}

public class B {
    @Autowired private final A a;

    public B(A a) {
        this.a = a;
    }
}
```

A와 B모두 필드 주입 기반이고, A를 만들려면 B가 필요하고, B를 만들려면 A가 필요해서 빈 생성 시점에 서로 무한 대기 상태가 발생합니다.

**생성자 주입은 빈을 만들기 "직전"에 모든 의존 객체를 찾아야 합니다.**
**순환 참조가 있으면 이 시점에서 바로 에러가 발생합니다.**
"필요한 객체를 아직 만들지도 않았는데, 생성자에서 달라고 하네?"
→ 스프링이 즉시 예외를 던집니다!

반면, **필드 주입은 빈 객체 생성 후 의존 객체를 주입합니다.**
**객체를 먼저 만들어 놓고, 나중에 필드에 값을 넣기 때문에 순환 참조가 늦게 드러납니다.** 실제 요청을 보낼 때까지 버그를 모르고 지나칠 수 있습니다.

# 싱글톤 레지스트리

싱글톤 레지스트리는 싱글톤 객체를 만들고 관리합니다. 그리고 스프링 IoC컨테이너도 싱글톤 레지스트리의 종류입니다. 즉 **빈을 싱글톤으로 만들고 관리**합니다.

**왜 빈을 싱글톤으로 만들까요?**

스프링이 주로 적용하는 대상은 자바 엔터프라이즈 기술을 사용하는 서버환경이고, 이 서버 환경에서는 서버 하나당 최대로 초당 수십에서 수백 번씩 브라우저나 여타 시스템으로 부터 요청을 받는 높은 성능이 요구됩니다.

그래서 **빈이 싱글톤으로 만들어지지 않았다면 클라이언트에서 요청이 올 때마다 오브젝트를 새로 만들어야해 서버가 감당하기 힘들 것**입니다. 그래서 싱글톤으로 빈을 만듭니다.

# IoC vs DI

많은 분들이 IoC와 DI를 같은 개념으로 혼동하기도 합니다.
하지만 둘은 밀접한 관련은 있지만 다른 개념입니다.

IoC(Inversion of Control, 제어의 역전)은 프로그램의 흐름 제어권이 바뀌는 구조 자체를 의미합니다.

DI(Dependency Injection, 의존성 주입)은 이러한 IoC 개념을 실제로 구현하는 방법 중 하나입니다. 즉, 객체가 필요로 하는 의존성을 외부에서 주입해주는 방식이죠.

쉽게 말해,
IoC는 원칙(설계 철학)이고
DI는 그 원칙을 실현하는 방법(기술)입니다.

스프링 프레임워크는 IoC 원칙을 따르며, 그 구현 방식으로 DI를 채택한 구조라고 볼 수 있습니다. 그래서 사람들은 스프링에서 IoC컨테이너를 DI컨테이너라고 부르기도 합니다.

# 참고문헌

참고

책: 이일민/토비스프링 vol 3.1/에이콘/2012.09.21

블로그: https://mangkyu.tistory.com/125