# Spring DI, IoC

558280-1
2026년 봄학기
4/9/2026
박경신

# Overview

- IoC (Inversion of Control)
- DI (Dependency Injection)
- Spring Container - BeanFactory, ApplicationContext
- Singleton Pattern - Bean
- Bean 등록
  - @ComponentScan, @Component, @Controller, @Service, @Repository (자동 등록)
  - @Configuration, @Bean (수동 등록)
- DI Types - field, setter, constructor
- Annotation DI - @Autowired (자동 주입), @Qualifier (선택)

# Before IoC

- Plain Java

```
private Encryptor enc = new Encryptor();
```

- 문제
  - 강한 결합 (tight coupling)
  - 구현 변경 시 전체 수정 필요
  - 테스트 어려움
- 핵심원인
  - "객체 생성 책임이 코드 내부에 묶여 있음"

# IoC (Inversion of Control)

- IoC (Inversion of Control) 제어의 역전
  - 프로그램의 제어를 다른 대상에게 맡기는 것임.
- Spring IoC
  - Spring Container가 Bean 생성, 관리, 의존 관계 주입 (DI) 담당함.
  - **Bean 생성부터 소멸까지의 생명주기 관리**를 개발자가 아닌 **컨테이너**가 대신 해줌.
  - Spring Container는 **ApplicationContext**이며, **IoC (Inversion of Control) 컨테이너** 혹은 **DI (Dependency Injection) 컨테이너**라고도 부름.

| | 생성 주체 |
|---|---|
| 기존 Java | 개발자가 객체 생성 |
| Spring | 컨테이너가 객체 생성 |

# Spring Container

- Spring Container 구현
  - **BeanFactory**
    - 스프링 컨테이너의 최상위 인터페이스이며, 스프링 Bean을 관리하고 조회하는 순수한 DI 역할을 담당함.
    - BeanFactory 계열의 인터페이스만 구현한 클래스는 단순히 컨테이너에서 객체를 생성하고 DI를 처리하는 기능만 제공함.
    - Factory Design Pattern을 구현한 것으로 BeanFactory는 빈을 생성하고 분배하는 책임을 지는 클래스.
    - Bean을 조회할 수 있는 **getBean() 메소드**가 정의되어 있음.
  - **ApplicationContext**
    - BeanFactory + 앱 개발에 필요한 편리한 부가기능 추가

# Spring Container

- 스프링 컨테이너는 객체(Bean) 인스턴스를 **Singleton** 으로 관리함
  - **스프링 컨테이너가 객체 1개만 생성함**
  - **스프링 컨테이너가 객체 (Bean)을 등록하고, 빈 조회 요청 시 새로 생성하지 않고 스프링 컨테이너에서 빈을 찾아서 반환함**
  - **스프링은 @Configuration이 붙은 클래스를 설정 정보로 사용함**
- Singleton pattern 은 객체를 하나만 생성하게 해서, 하나의 객체에서 처리하게 함.

# Classical Singleton Pattern

```
public class Singleton {
  // static field containing its only instance
  private static Singleton uniqueInstance;

  // private default constructor
  private Singleton() { }

  // static factory method for obtaining the
instance
  public static Singleton getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
    }
    return uniqueInstance;
  }
} // classical implementation
Singleton singleton = Singleton.getInstance();
```

# 스프링 컨테이너 생성, 빈 등록, 의존관계 설정

1. 스프링은 스프링 컨테이너를 생성하고,
2. 스프링 컨테이너는 Key=빈 이름, Value=빈 객체 형태로 빈을 저장,
   - Key (**빈 이름**)은 **메서드 이름**으로 사용, 실제 반환하는 객체를 Value (**빈 객체**)에 저장함
3. 스프링 컨테이너는 설정 정보 (**Configuration**)을 참고해서 의존 관계 주입 (**Dependency Injection**) 함.

```
@Configuration
        ↓
Container
        ↓
[Bean1, Bean2, Bean3]
        ↓
DI 연결
```

# DI (Dependency Injection)

- DI (Dependency Injection) 의존 주입
  - 객체 간의 **의존**을 **외부에서 주입하는 것**을 말함.
  - new를 없애는 설계 방식
- DI (Dependency Injection) 는 IoC를 구현하는 방법
  - 의존하는 객체를 직접 생성하는 대신, 의존 객체를 전달받는 방식을 사용함.

```
// Plain Java
MemberService -> new MemoryRepository();

// DI
MemberService <- MemberRepository
```

# Dependency (의존성)

- Dependency (의존성)
  - 한 객체가 다른 객체를 사용하는 관계

```
// Service는 Repository 없이는 동작할 수 없음 (즉, 의존)
MemberService -> new MemoryRepository();
```

- Spring 생산/유지보수 용이성의 핵심요소
  - Spring이 유지보수하기 쉬운 이유는 "의존성 관리" 때문
- Single class > Dependency per code > Dependency Injection
  - Single class 하나의 클래스에 모든 기능을 넣어서 의존성이 큼
  - Dependency per code 클래스 간 의존성이 증가 (서로 연결됨)
  - Dependency Injection 의존성을 외부에서 관리

# Single class 의존성

- 하나의 클래스에 모든 기능을 다 집어넣는 경우
  - 4만 줄 짜리 클래스 본 적 없죠?
- 사소한 변경 하나에 의해서도 전체 코드를 수정하는 결과
  - 편집의 어려움
  - 비효율적 재사용
  - 이 클래스에 의존하고 있는 다른 클래스들이 변경에 취약해짐

# 코드 레벨 의존성의 문제

- 일반적인 자바 개발에서 이뤄지는 코드 구성의 결과

```
private Encryptor enc = new Encryptor();
```

- 이 코드가 정상적으로 작동하기 위해서는
  - Encryptor 타입의 클래스 Encryptor가 존재해야 할 것
- 만약 Encryptor 대신 그 서브타입인 FastEncryptor를 사용해야 한다면?
  - 내 코드를 수정해야 함

```
private Encryptor enc = new FastEncryptor();
```

  - Encryptor를 사용하던 클래스가 많으면 많을수록 더 많은 수정
    - 더 많은 테스트
    - 더 많은 오류
    - 더 많은 수정 전파
- Encryptor가 완성될 때 까지 내 코드를 테스트 불가능

# 조금 더 나은 모델

- 외부 코드에 의한 타입 전달

```
public class  FileEncryptor{
        private Encryptor enc;

        public FileEncryptor(Encryptor enc) {
                this.enc = enc;
        }
```

- 생성자 호출시 Encryptor 타입의 아무 클래스나 존재하면 가능
  - 다양한 구현체 사용 가능
  - 코드 수정 없이 교체 가능
- 전통적인 code with interface 모델

# Dependency Injection (의존성 주입)

- 위 모델을 사용하기 위한 코드

```
Encryptor enc = new Encryptor();
FileEncryptor fileEnc = new FileEncryptor(enc);
```

- FileEncryptor가 의존하는 객체는 자신의 코드가 아닌 외부에서 생성되어 넣어 줌
  - 이러한 방식을 **의존성 주입 (Dependency Injection)**이라 부름

# 의존성 주입 방식

- Factory 패턴 (객체 생성 역할 분리)
  - 한 타입(Encryptor)을 만족시키는 다양한 서브타입(FastEncryptor, PlainEncryptor, …)을 생성해서 돌려주는 코드

```
Enc = EncFactory.getEncryptor(EncType.Fast);
```

EncFactory

```
if (type == EncType.Fast){
        return new FastEncryptor();
}
```

  - 변경이 일어나면 이 조립기의 코드에만 영향이 미침
    - 팩토리에 Fast 대신 SuperFast Encryptor를 쓰는 코드만 넣으면 다른 코드에는 영향이 없음

# 의존성 주입 방식

- 또 다른 장점
  - Encryptor 클래스가 완성되지 않았다면
    - 팩토리에 MockEncryptor 임시 클래스를 장착
    - 해당 클래스는 가짜 객체로, 무조건 일정한 결과를 돌려주는 단순한 코드
    - 이러한 클래스를 MockObject 혹은 test stub이라 부름
  - Encryptor 서브클래스들과 FileEncryptor 클래스를 작업하는 사람이 동시에 작업 분담 가능

# DI 방식

- Field DI
  - 간단하지만 비권장
- Setter DI
  - 선택적 의존성
- Constructor DI (추천)
  - **final** 가능
  - 안정성 높음

# 생성자 방식 DI

- 생성자를 통해 **생성 시** 의존 객체를 전달받음

```
public class FileEncryptor{
        private Encryptor enc;

        public FileEncryptor(Encryptor  enc) {
                this.enc = enc;
        }
… }
```

- 생성자 실행 후 언제나 객체 사용이 가능

# Setter 방식 DI

- **생성 후** setter로 의존 객체를 설정 (선택적 의존성)
- 자바빈즈 영향으로 setPropertyName()과 같은 메소드로 속성 설정 가능

```
public class FileEncryptor{
        private Encryptor encryptor;

        public void setEncryptor(Encryptor enc) {
                this.encryptor = enc;
        }
... }
```

- 메소드 이름으로 속성의 타입을 추측 가능

# Field 방식 DI

- **필드에 직접 주입**

```
@Component
public class FileEncryptor {
        @Autowired
        private Encryptor encryptor;
... }
```

# Spring Bean

- Spring Bean
  - Container 가 관리하는 순수 자바 객체(POJO)
  - Container 는 설정을 읽은 후 Spring Bean 객체를 생성하고, 서로 의존성 있는 Bean 객체들을 주입(DI)함
  - Container 는 Bean 생성부터 주입, 소멸하는 전체 과정 (즉, Spring Bean Life Cycle) 을 관리함

# Spring 빈 등록

- Bean 등록
  - 자동 등록 **@ComponentScan**으로 스캔되어 Bean으로 생성
    - **@Component** 일반 Bean
    - **@Controller** 웹 요청 처리 Bean
    - **@Service** 비즈니스 로직 Bean
    - **@Repository** 데이터 접근 Bean
  - Auto-configuration (자동 설정)
    - **@EnableAutoConfiguration**은 프로젝트의 **외부 라이브러리들의 빈들을 등록함**
  - 수동 등록 User-defined Configuration
    - Container 가 로딩하는 설정(**@Configuration**) 파일에 **@Bean** 을 사용하여 정의

# Spring 빈 등록

- **@ComponentScan** 를 통한 **자동** 빈 등록
  - @ComponentScan은 **@Component**가 붙은 객체를 찾아 자동으로 빈 등록하는 방법이 있음.
  - **@Controller, @Service, @Repository**는 모두 @Component를 포함하고 있으며, 해당 어노테이션으로 등록된 클래스들은 스프링 컨테이너에 의해 자동으로 생성되어 스프링 빈으로 등록됨.
  - 일반적으로 Bean을 생성하기 위해서는 @Component가 더 많이 사용됨

# Spring 빈 등록

- **@Configuration, @Bean**을 통한 **직접** 빈 등록
  - 스프링이 뜰 때에 스프링은 자동으로 @Configuration 이 붙은 클래스(예를 들어, AppConfig)를 찾아서 구성 정보로 사용함
    - 클래스에 @Configuration 을 달면 해당 클래스가 **JavaConfig**에서 Bean을 정의하는 소스로 사용된다는 것을 나타냄
    - @Configuration 주석이 달린 클래스를 하나만 사용할 수도 있고 **여러 개 사용할 수도 있음**
    - @Configuration은 XML의 <beans/> 요소와 동등함
  - @Configuration 구성 정보에 @Bean을 통해 스프링 컨테이너에 직접 빈을 등록하고 의존 관계 주입을 처리할 수 있음
  - 빈 객체로 등록하고 싶은 **메서드 위에 @Bean 추가**

# @Bean

- **@Bean [method]**
  - @Configuration 설정된 클래스의 메소드에서 사용 가능
  - 메소드의 리턴 객체가 스프링 빈 객체임을 선언함
  - **빈의 이름**은 기본적으로 **메소드의 이름**
  - @Bean(name="name")으로 이름 변경 가능
  - @Scope를 통해 객체 생성을 조정할 수 있음

# @Configuration, @Bean

- @Configuration 해당 클래스가 스프링 설정임을 나타냄
- @Bean 해당 메소드가 빈(Bean) 객체를 만들어 냄을 의미

```
@Configuration
public class CoffeeConfig {
    // Configuring origin for Coffee
    @Bean // 메소드 이름이 빈 객체 이름임 origin
    public Origin origin() {
        return new Origin("Costa Rica");
    }
    // Configuring roast for Coffee
    @Bean // 메소드 이름이 빈 객체 이름임 roast
    public Roast roast() {
        return new Roast(2);
    }
}
```

# @Configuration, @Bean

- @Configuration은 @SpringBootApplication 내에서도 제공됨

```
@SpringBootApplication
//@ComponentScan
//@EnableAutoConfiguration
//@SpringBootConfiguration // ==> wraps @Configuration
public class SpringBootHelloControllerApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootHelloControllerApplication.class, args);
    }
    @Bean
    public Hello hello() {
        return new Hello("Application @Bean says Hello");
    }
}
```

# Bean Visibility

## AppConfig.java

```
@Configuration
public abstract class VisibilityConfiguration {
  @Bean
  public Bean publicBean() {
     Bean bean = new Bean();
     bean.setDependency(hiddenBean());
     return bean;
  }
  @Bean
  protected HiddenBean hiddenBean() {
     return new Bean("protected bean");
  }
  @Bean
  HiddenBean secretBean() {
     Bean bean = new Bean("package-private bean");
     // hidden beans can access beans defined in the 'owning' context
     bean.setDependency(outsideBean());
  }
  @ExternalBean
  public abstract Bean outsideBean()
}
```

https://docs.spring.io/spring-javaconfig/docs/1.0.0.m3/reference/html/creating-bean-definitions.html

# Bean Visibility

- 다음 appConfig.xml과 일치함

```
<beans>
 <!-- the configuration above -->
 <bean class="my.java.config.VisibilityConfiguration"/>

 <!-- Java Configuration post processor -->
 <bean
class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

 <bean id="mainBean" class="my.company.Bean">
    <!-- this will work -->
    <property name="dependency" ref="publicBean"/>
    <!-- this will *not* work -->
    <property name="anotherDependency" ref="hiddenBean"/>
 </bean>
</beans>
```

https://docs.spring.io/spring-javaconfig/docs/1.0.0.m3/reference/html/creating-bean-definitions.html

# Annotation 기반 의존 관계 주입

- **@Autowired, @Qualifier**를 통한 DI(의존성 주입)
  - Annotation 기반 Dependency Injection (의존성 주입)은 의존성 대상 객체를 주입받을 클래스 내부의 객체를 저장할 변수에 @Autowired 와 @Qualifier 를 조합하여 정의함
  - Container 는 두 annotation의 속성 값을 파악하여 적절한 의존성 대상 객체를 찾아 주입함
    - 의존성 주입 시 의존성을 주입 받을 객체와 의존성 대상 객체 모두 Spring Bean 객체이어야 함

# @Autowired 의존관계 주입

- @Autowired를 통한 DI(의존성 주입)
  - 스프링 컨테이너에 빈들을 모두 등록한 후에 DI(의존성 주입)
  - @Autowired는 기존에 XML에 <property>, <constructor-arg>를 통해 **DI** 해오던 방식을 **자동**으로 해주는 Annotation
  - @Autowired 의존성 주입 방법은 다음 3가지 방식으로 정의
    1. **Field 방식 – 가장 기본적인 방식**
    2. **Setter 방식**
    3. **Constructor 방식 – 추천하는 방식**

# @Autowired Field 방식

- 필드(field) 기반 DI는 가장 기본적인 형식이며 @Autowired를 이용하여 간단하게 의존주입(객체생성)
  - 필드에 final 를 사용할 수 없음 – 불변성(immutable) 허용하지 않음
  - 사용이 간단하여 의존성 주입 대상이 많아질 수 있음
  - 의존관계가 가려짐

```
@Component
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Car {
    // field-based dependency injection (cannot use final)
    @Autowired
    private Engine engine; // 스프링이 engine 객체를 생성해서 주입
    // field-based dependency injection (cannot use final)
    @Autowired
    private Transmission transmission; // 스프링이 transmission 객체를 생성해서
주입
}
```

# @Autowired Field 방식

```
@Configuration
public class CarConfig {
    // Configuring Engine for Car
    @Bean
    public Engine engine() {
        return new Engine("VR6", 6);
    }
    // Configuring Transmission for Car
    @Bean
    public Transmission transmission() {
        return new Transmission("Dual Clutch");
    }
}
```

# @Autowired Setter 방식

- @Autowired를 setter 메서드에 등록해서 사용함
  - 필드에 final 를 사용할 수 없음
  - 선택적이고 변화 가능한 의존관계에 사용함

```
@Component
@Data
@NoArgsConstructor
public class Car {
    private Engine engine;
    private Transmission transmission;
    // setter-based dependency injection
    @Autowired // setter 매개변수에 스프링이 자동으로 객체를 생성해서 주입
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
    // setter-based dependency injection
    @Autowired // setter 매개변수에 스프링이 자동으로 객체를 생성해서 주입
    public void setTransmission(Transmission transmission) {
        this.transmission = transmission;
    }
```

# @Autowired Setter 방식

- 속성 입력 방식
  - 자바 빈즈의 setter 직접 호출

```
@Configuration
public class CarConfig {
    @Bean
    public Engine engine() {
        return new Engine("v5", 3);
    }
    @Bean
    public Transmission transmission() {
        return new Transmission("automatic");
    }
    @Bean
    public Car car() {
        Car car = new Car();
        car.setEngine(engine()); // setter DI
        car.setTransmission(transmission()); // setter DI
        return car;
    }
}
```

# @Autowired Constructor 방식

- 생성자(constructor) 기반 DI는 생성자에 @Autowired를 사용하여 의존주입(객체생성)
  - **final 선언이 가능**함으로 객체를 재할당하는 것을 방지할 수 있음
  - 생성자가 호출될 때 딱 한 번 주입됨. 순환 참조가 방지됨
  - 생성자가 하나일 경우 @Autowired를 생략할 수 있음
  - 필수 의존성 주입에 유용함

```
@Component
@Data
public class Car {
    private final Engine engine;
    private final Transmission transmission;
    // constructor-based dependency injection
    @Autowired // 스프링이 생성자 매개변수에 객체를 생성해서 주입
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

# @Autowired Constructor 방식

- 생성자 방식
  - 직접 생성자 호출하면서 매개변수 입력

```
@Configuration
public class CarConfig {
    // Configuring Engine for Car
    @Bean
    public Engine engine() {
        return new Engine("v8", 5); // constructor DI
    }

    // Configuring Transmission for Car
    @Bean
    public Transmission transmission() {
        return new Transmission("sliding"); // constructor DI
    }
}
```

# @RequiredArgsContructor

- @RequiredArgsContructor
  - Lombok을 사용하여 간단한 방법으로 생성자 기반 DI 가능
  - **@RequiredArgsConstructor**는 **final 필드**나, **@NonNull 이 붙은 필드**에 대해 생성자를 자동 생성
  - 새로운 필드를 추가할 때 다시 생성자를 만들어서 관리해야하는 번거로움을 없애줌

```
@Component
@Data
@RequredArgsConstructor
public class Car {
    private final Engine engine;
    private final Transmission transmission;
}
```

# list, map, set

- 각각 자바의 List, Map, Set 컬렉션 타입에 대응 - **List**

```
@Repository
@Data
public class ChocolateRepository {
    @Autowired
    private List<String> nameList; // field DI 방식
    public void printNameList() {
        System.out.println(Arrays.toString(nameList.toArray()));
    }
}

@Configuration
public class ChocolateConfig {
    // Configuring nameList for ChocolateRepository
    @Bean
    public List<String> nameList() {
        return Arrays.asList("Lindt", "Godiva", "Ghirardelli");
    }
}
```

# list, map, set

- 각각 자바의 List, Map, Set 컬렉션 타입에 대응 - **Set**

```
@Repository
@Data
public class ChocolateRepository {
    @Autowired
    private Set<String> nameSet; // field DI 방식
    public void printNameSet() {
        System.out.println(Arrays.toString(nameSet.toArray()));
    }
}

@Configuration
public class ChocolateConfig {
    // Configuring nameSet for ChocolateRepository
    @Bean
    public Set<String> nameSet() {
        return new HashSet<>(Arrays.asList("Milk Chocolate", "Dark Chocolate",
"Pave Chocolate"));    }
}
```

# list, map, set

- 각각 자바의 List, Map, Set 컬렉션 타입에 대응 - **Map**

```
@Repository
@Data
public class ChocolateRepository {
    @Autowired
    private Map<Integer, String> nameMap; // field DI 방식
    public void printNameMap() {
        nameMap.entrySet().stream().forEach(e-> System.out.println(e));
    }
}
```

# list, map, set

- 각각 자바의 List, Map, Set 컬렉션 타입에 대응 - **Map**

```
@Configuration
public class ChocolateConfig {
    // Configuring nameMap for ChocolateRepository
    @Bean
    public Map<Integer, String> nameMap() {
        Map<Integer, String>  nameMap = new HashMap<>();
        nameMap.put(1, "M&M's");
        nameMap.put(2, "Reese's");
        nameMap.put(3, "Hershey");
        return nameMap;
    }
}
```

# @Qualifier

- **@Qualifier**는 메서드, 파라메터, 필드, 애너테이션에 정의 가능함
  - value 속성값은 Spring Bean 의 이름이 됨
- @Qualifier는 사용할 의존 객체를 **선택**할 수 있게 해줌
  - @Qualifier는 클래스 타입은 같지만 이름이 다른 여러 Spring Bean 이 있을 경우 이 중 정의된 이름의 Spring Bean 을 주입 받기 위해 사용함
  - @Qualifier가 아닌 **@Primary**로도 처리 가능함

# @Qualifier

- 다음 두 단계를 설정해주어야 함
  1. Configuration 에서 빈의 **한정자(@Qualifier) 이름**을 설정
  2. @Autowired 주입 대상(field, setter, constructor)에 **@Qualifier를 설정** (이때 Configuration 에서 설정한 한정자 이름을 사용)
     - 여기서 주의할 점은 한정자 이름이 일치하지 않을 시 객체가 존재하지 않아 Exception 발생

```
@Repository
@Data
public class ChocolateRepository {
    @Autowired
    @Qualifier("chocolateList") // 주입된 bean 이름을 chocolateList로 설정
    private List<Chocolate> chocolateList;
    public void printChocolateList() {
        chocolateList.forEach(System.out::println);
    }
}
```

# @Qualifier

```
@Configuration
public class ChocolateConfig {
    @Bean
    @Qualifier("chocolateList")
    public List<Chocolate> chocolateList() {
        List<Chocolate> chocolateList = new ArrayList<Chocolate>();
        chocolateList.add(new Chocolate(new Brand("Lindt", "Swiss"), new Type("Milk
Chocolate")));
        chocolateList.add(new Chocolate(new Brand("Godiva", "Belgium"), new
Type("Dark Chocolate")));
        chocolateList.add(new Chocolate(new Brand("Ghirardelli", "USA"), new
Type("Dark Chocolate")));
        chocolateList.add(new Chocolate(new Brand("Ferrero Rocher", "Italy"), new
Type("Nutella Chocolate")));
        chocolateList.add(new Chocolate(new Brand("Royce", "Japan"), new
Type("Pave Chocolate")));
        return chocolateList;
    }
}
```

# @Primary

- **@Primary** 를 통한 우선 Bean 지정
  - 같은 타입의 Spring Bean 이 여러 개 존재할 때, **@Primary**가 붙은 빈을 우선적으로 주입함
- 주입 받는 쪽에서 @Qualifier 를 별도로 지정하지 않아도 됨 – **코드가 간결해짐**
- @Qualifier 와 동시에 사용할 경우 @Qualifier가 우선함
- 같은 타입에 **@Primary**가 둘 이상이면 NonUniqueBeanDefinitionException 발생

# @Primary

```
@Configuration
public class ChocolateConfig {
    @Bean
    @Primary // 우선 주입될 빈
    public List<Chocolate> primaryChocolateList() {
        return Arrays.asList(new Chocolate(new Brand("Lindt", "Swiss"), new
Type("Milk Chocolate")), new Chocolate(new Brand("Godiva", "Belgium"), new
Type("Dark Chocolate")));
    }
    @Bean
    public List<Chocolate> otherChocolateList() {
        …
    }
}
@Repository
public class ChocolateRepository {
    @Autowired
    //@Qualifier 없이도 @Primary 빈 ("primaryChocolateList") 자동 주입
    private List<Chocolate> chocolateList;
}
```

# LoginUser 자바빈

- LoginUser 클래스

```
@Component // Spring Component
@Data // Lombok @Getter/@Setter/@ToString
@NoArgsContructor // Lombok Default Constructor
@AllArgsConstructor // Lombok Parameterized Constructor
public class LoginUser {
    private String loginId;
    private int password;
}
```

# LoginUserRepository 자바빈

- LoginUserRepository 클래스

```
@Repository // Spring Component
@Data // Lombok @Getter/@Setter/@ToString
public class LoginUserRepository {
    List<LoginUser> userList;

    public void printList() {
       System.out.println("Repository printList userList=" + this.userList);
    }
}
```

# LoginUserService 자바빈

- LoginUserService 클래스

```
@Service // Spring Component
@NoArgsConstructor
@Data
public class LoginUserService {
    private LoginUserRepository userRepository;
    // setter-based dependency injection
    @Autowired
    public void setUserRepository(LoginUserRepository userRepository) {
        this.userRepository = userRepository;
    }
    public List<LoginUser> findAll() {
        return this.userRepository.getUserList();
    }
}
```

# LoginUserController 자바빈

- LoginUserController 클래스

```
@Controller // Spring Component
@RequestMapping("/user")
public class LoginUserController {
    private final LoginUserService userService;
    // constructor-based dependency injection
    @Autowired // 생성자가 1개만 있으면 생략 가능
    public LoginUserController(LoginUserService userService) {
        this.userService = userService;
    }
    @GetMapping("/list") // localhost:8080/di/user/list
    public String list(Model model) {
        model.addAttribute("userList",
this.userService.getUserRepository().getUserList());
        return "userlist"; // templates/userlist.html
    }
}
```

# LoginUserConfig에서 @Bean 주입

```
@Configuration
public class LoginUserConfig {
    @Bean
    protected List<LoginUser> userList() {
        List<LoginUser> userList = new ArrayList<>();
        userList.add(new LoginUser("Kim", 12345));
        userList.add(new LoginUser("Lee", 6789));
        userList.add(new LoginUser("Park", 321));
        userList.add(new LoginUser("DIS", 98765));
        return userList;
    }
    @Bean
    public LoginUserRepository userRepository() {
        LoginUserRepository userRepository = new LoginUserRepository();
        userRepository.setUserList(userList());
        return userRepository;
    }
}
```

# 메인에서 Bean 등록 확인 테스트

- Main코드에서 ApplicationContext의 getBean 사용

```
public static void main(String[] args) {
    ConfigurableApplicationContext appContext =
                                        SpringApplication.run(SpringBootDiApplication.class, args);
    Car car = appContext.getBean(Car.class);
    System.out.println("car=" + car);
    Coffee coffee = appContext.getBean(Coffee.class);
    coffee.print();
    ChocolateRepository chocolateRepository =
                                    appContext.getBean(ChocolateRepository.class);
    chocolateRepository.printChocolate();
    chocolateRepository.printNameList();
    chocolateRepository.printNameSet();
    chocolateRepository.printNameMap();
    chocolateRepository.printChocolateList();
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public interface Client {
    void doSomething();
}

@Component("client1") // 빈 이름 client1는 ClientA 객체
public class ClientA implements Client {
    @Autowired
    private Service service1; // client1 <- service1
    @Override
    public void doSomething() {
        System.out.println("ClientA: " + service1.getInfo());
    }
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public class ClientB implements Client {
    Service service;
    public ClientB(Service service) {
        this.service = service;
    }
    @Override
    public void doSomething() {
        System.out.println("ClientB: " + service.getInfo() + " " + service.getInfo());
    }
}
public class ClientC implements Client {
    Service service;
    public ClientC(Service service) {
        this.service = service;
    }
    @Override
    public void doSomething() {
        System.out.println("ClientC: " + service.getInfo() + " " + service.getInfo() + " " +
service.getInfo());
    }
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
public interface Service {
    String getInfo();
}

@Component("service1") // 빈 이름 service1은 ServiceE 객체
public class ServiceE implements Service {

    @Override
    public String getInfo() {
        return "ServiceE's Info";
    }

}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
@Component("service2") // 빈 이름 service2는 ServiceF 객체
public class ServiceF implements Service {

    @Override
    public String getInfo() {
        return "ServiceF's ServiceF's Info";
    }

}
public class ServiceG implements Service {

    @Override
    public String getInfo() {
        return "ServiceG's ServiceG's ServiceG's Info";
    }

}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

```
@Configuration
public class AppConfig {
    @Bean("service3") // 빈 이름 service3는 ServiceG 객체
    public Service getService3() {
        return new ServiceG();
    }
    @Bean("client2") // client2 (ClientB) <- service2 (ServiceF)
    public Client getClient3(@Qualifier("service2") Service service2) {
        return new ClientB(service2);
    }
    @Bean("client3") // client3 (ClientC) <- service3 (ServiceG)
    public Client getClient4(@Qualifier("service3") Service service3) {
        return new ClientC(service3);
    }
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

- 메인자바 테스트

```
public static void main(String[] args) {
    ConfigurableApplicationContext appContext =
                            SpringApplication.run(SpringBootDiApplication.class, args);
    Client client1 = (Client)appContext.getBean("client1");
    client1.doSomething(); // client1 <- service1
    Client client2 = (Client)appContext.getBean("client2");
    client2.doSomething(); // client2 <- service2
    Client client3 = (Client)appContext.getBean("client3");
    client3.doSomething(); // client3 <- service3
}
```

# 같은 인터페이스를 구현한 Bean을 구분하여 의존성 주입

- 실행 결과

```
ClientA: ServiceE's Info
ClientB: ServiceF's ServiceF's Info ServiceF's ServiceF's Info
ClientC: ServiceG's ServiceG's ServiceG's Info ServiceG's
ServiceG's ServiceG's Info ServiceG's ServiceG's ServiceG's Info
```