Spring

Spring

- Spring Framework
  - ✓ 자바 엔터프라이즈 애플리케이션 개발을 편하게 해주는 오픈 소스로 경량급 프레임워크
  - ✓ 특징
    - 복잡함에 반기를 들어서 나온 프레임워크
    - 검증된 Architecture 의 재사용과 일관성 유지
    - 빠른 구현 시간 – 쉬운 관리
    - 스프링은 IoC(Inversion of Control - 제어의 역전), DI(Dependency Injection – 의존성 주입), AOP(Aspect Oriented Programming – 관점지향 프로그래밍) 를 지원
    - 편리한 MVC 구조를 가지고 있고 WAS에 종속되지 않는 개발 환경
    - 영속성 - MyBatis, Hibernate, JPA등과 같은 프레임워크 와 의 연동을 지원
    - 트랜잭션 처리를 위한 일관된 방법을 제공
    - Cloud에 맞춰진 마이크로 서비스 개발이 용이
    - 확장성과 향상된 성능을 제공하는 Reactive 프로그래밍에 적합
    - Spring Security, Integration, Batch, Cloud 지원

Spring

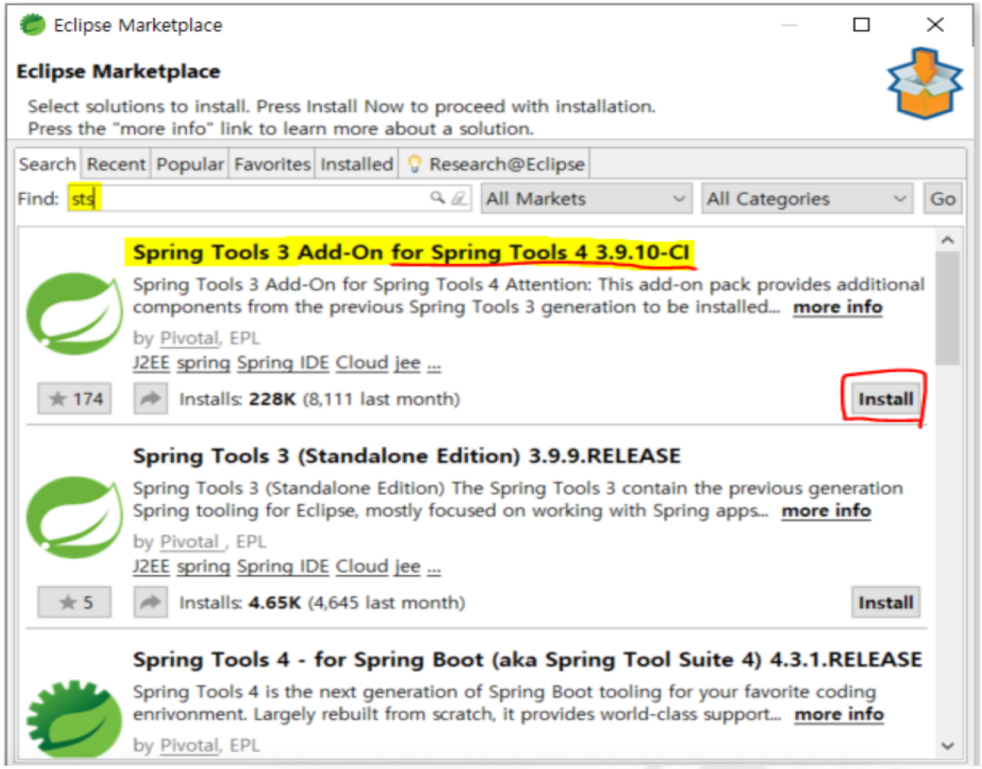
- ❖ 개발 환경
  - ✓ JDK SE 설치
    - Spring 4.0 이상 사용시에는 1.6 이상
    - Spring 5.0 이상 사용시 에는 1.8 이상
    - 최신의 IDE 에서는 1.11 이상
  - ✓ Web Programming을 하기 위해서는 WAS(Tomcat 8.0 이상 – Spring Boot 에서는 내장)
  - ✓ 스프링 IDE
    - Eclipse Plugin 이용(Spring Tool Suite)
    - Spring 전용 Eclipse
      - ◆ STS(Spring Tool Suite – Spring 전용 Eclipse) 사용
        - https://spring.io/tools : 최신 버전
        - https://github.com/spring-projects/toolsuite-distribution/wiki/Spring-Tool-Suite-3 : 모든 버전 – Spring Legacy Project를 사용하기 위해서 3.x버전 사용
      - 프레임워크 전용 Eclipse 이용: 전자 정부 프레임워크, AnyFramework(삼성) 등
    - Intellij 사용

Spring

❖ 개발 환경

✓ STS4에서 Spring Legacy Project 사용을 위한 설정

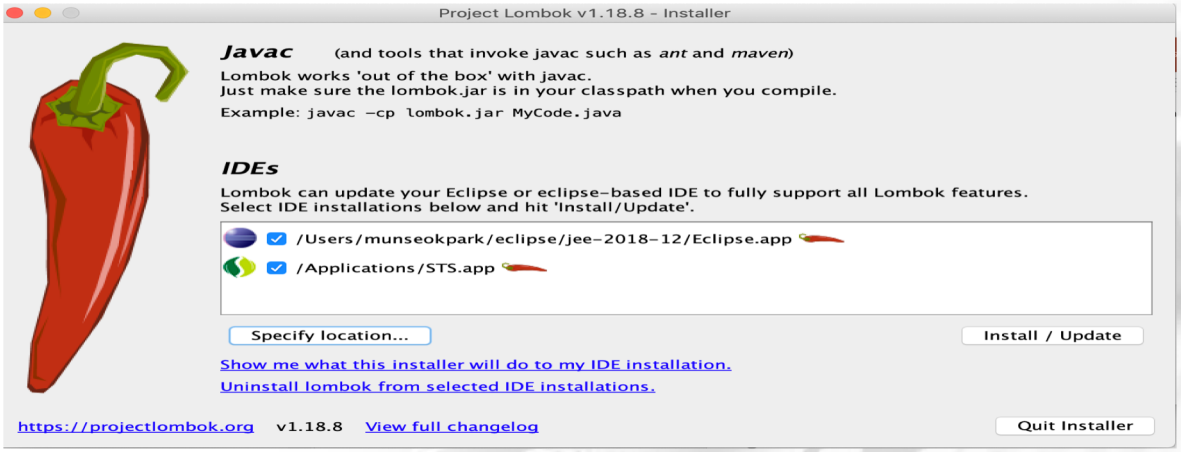
- 메뉴에서 [Help] – [Eclipse MarketPlace] 실행
- STS 검색
- Spring Tools 3 Add-On for Spring Tools 4 옆에 있는 install 버튼 클릭



lombok

- lombok
  - ✓ lombok
    - 반복적인 Getter/Setter, ToString과 같은 반복적인 자바 코드를 컴파일할 때 자동으로 생성해주는 라이브러리
  - ✓ 설치
    - https://projectlombok.org/download 에서 jar 파일 다운로드
    - 터미널에서 다운로드 받은 파일 실행

      java -jar lombok.jar
    - 사용할 IDE를 선택하고 install하고 IDE 재시작 – Mac 의 경우 contents 디렉토리 안의 Eclipse 디렉토리 안의 ini 파일 선택



lombok

- lombok
  - ✓ lombok 어노테이션
    - 생성자 자동 생성
      - Lombok을 사용하면 생성자도 자동으로 생성
      - @NoArgsConstructor 어노테이션은 파라미터가 없는 기본 생성자를 생성
      - @AllArgsConstructor 어노테이션은 모든 필드 값을 파라미터로 받는 생성자를 생성
      - @RequiredArgsConstructor 어노테이션은 final이나 @NonNull인 필드 값만 파라미터로 받는 생성자를 생성
    - @Builder
      - 빌더 패턴을 이용한 인스턴스 생성
      - 클래스이름.builder().속성이름(값)...build() 메서드로 인스턴스 생성

lombok

- lombok
  - ✓ lombok 어노테이션
    - 접근자/설정자 자동 생성
      - @Getter와 @Setter
      - xxx라는 필드에 선언하면 자동으로 getXxx()(boolean 타입인 경우, isXxx())와 setXxx() 메서드를 생성
      - 클래스 레벨에 @Getter 또는 @Setter를 선언해줄 경우 모든 필드에 접근자와 설정자가 자동으로 생성
    - @ToString
      - toString 메서드 자동 생성
      - @ToString(exclude三{"변수명"}): 원하지 않는 속성을 제외한 toString() 메서드 생성
    - @EqualsAndHashCode: equals 메서드와 hashCode 메서드 자동 생성
    - @NotNull: 해당 변수가 null 체크. NullPointerException 예외 발생
    - @Log: log 변수 자동 생성
    - @Value: 설정 내용을 호출해서 값을 대입
    - @Data: @Getter, @Setter, @RequiredArgsConstructor, @ToString, @EqualsAndHashCode을 한꺼번에 설정해주는 어노테이션

lombok

- lombok
  - ✓ lombok 의존성 설정
    - pom.xml

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.24</version>
    <scope>provided</scope>
</dependency>
```

    - build.gradle

```
compileOnly group: 'org.projectlombok', name: 'lombok', version: '1.18.24'
```

Spring

- 스프링 프로젝트 종류
  - ✓ Spring Boot(Spring Starter Project): 간단하게 실행 및 배포가 가능한 수준의 애플리케이션을 만들 때 사용하는데 WAS 설정없이 실행이 가능하기 때문에 테스트 하기에 편리하지만 기존의 웹 프로젝트 설정과 다른 방법으로 작성해야 하고 JSP 설정은 별도
  - ✓ Spring Template Project(Spring Legacy Project): WAS를 사용하거나 이전에 Spring Project를 만들어 본 경우 사용하는 방식인데 WAS로 인한 리소스 소모가 심하다는 단점이 있지만 기존 프로젝트들이 이 방식으로 많이 만들어져 있음

Spring

❖ STS에서 생성한 프로젝트의 pom.xml의 스프링 버전 설정

```
<properties>
    <java-version>1.8</java-version>
    <org.springframework-version>5.0.7.RELEASE</org.springframework-version>
    <org.aspectj-version>1.6.10</org.aspectj-version>
    <org.slf4j-version>1.6.6</org.slf4j-version>
</properties>
```

✓ <org.springframework-version> 태그

- 스프링 모듈의 버전을 설정하기 위한 속성 설정
- 프로젝트 종류에 따라 다르게 이름이 다르게 설정되어 있음
- 사용하고자 하는 스프링 버전으로 변경

Spring

- Spring Java Application Project
  - ✓ STS를 실행시켜 [File] – [New] – [Spring Legacy Project]를 선택하고 Simple Spring Maven 프로젝트를 생성 – SpringIOCAndDI
  - ✓ pom.xml 파일에서 Java는 1.8 Spring은 5.0.7 버전으로 설정하고 마우스 오른쪽을 클릭하고 [Maven] – [Update Project]를 실행

```
<properties>
    <!-- Generic properties -->
    <java.version>1.8</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <!-- Spring -->
    <spring-framework.version>5.0.7.RELEASE</spring-framework.version>
    <!-- Hibernate / JPA -->
    <hibernate.version>4.2.1.Final</hibernate.version>
    <!-- Logging -->
    <logback.version>1.0.13</logback.version>
    <slf4j.version>1.7.5</slf4j.version>
    <!-- Test -->
    <junit.version>4.12</junit.version>
</properties>
```

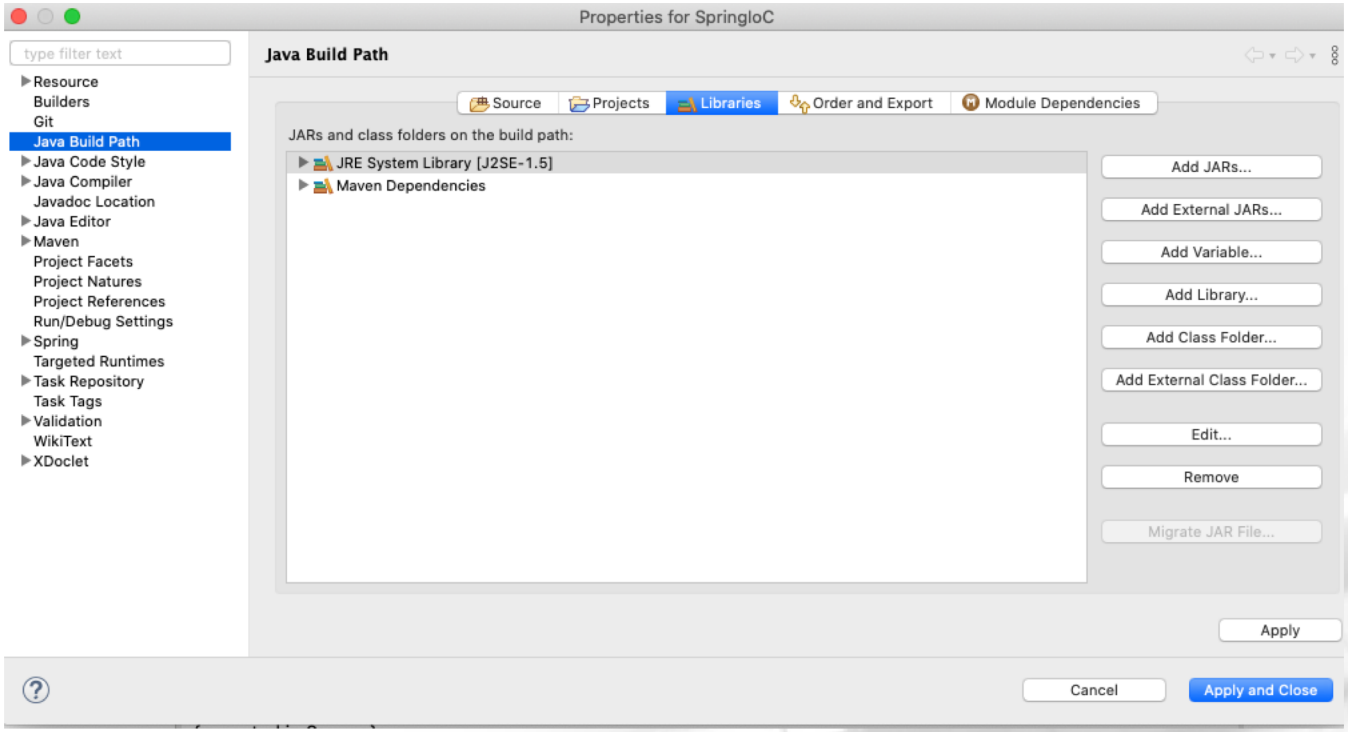
Spring

- Spring Java Application Project
  - ✓ pom.xml 파일에 lombok 라이브러리를 사용하기 위한 의존성을 설정

```
<dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.24</version>
</dependency>
```

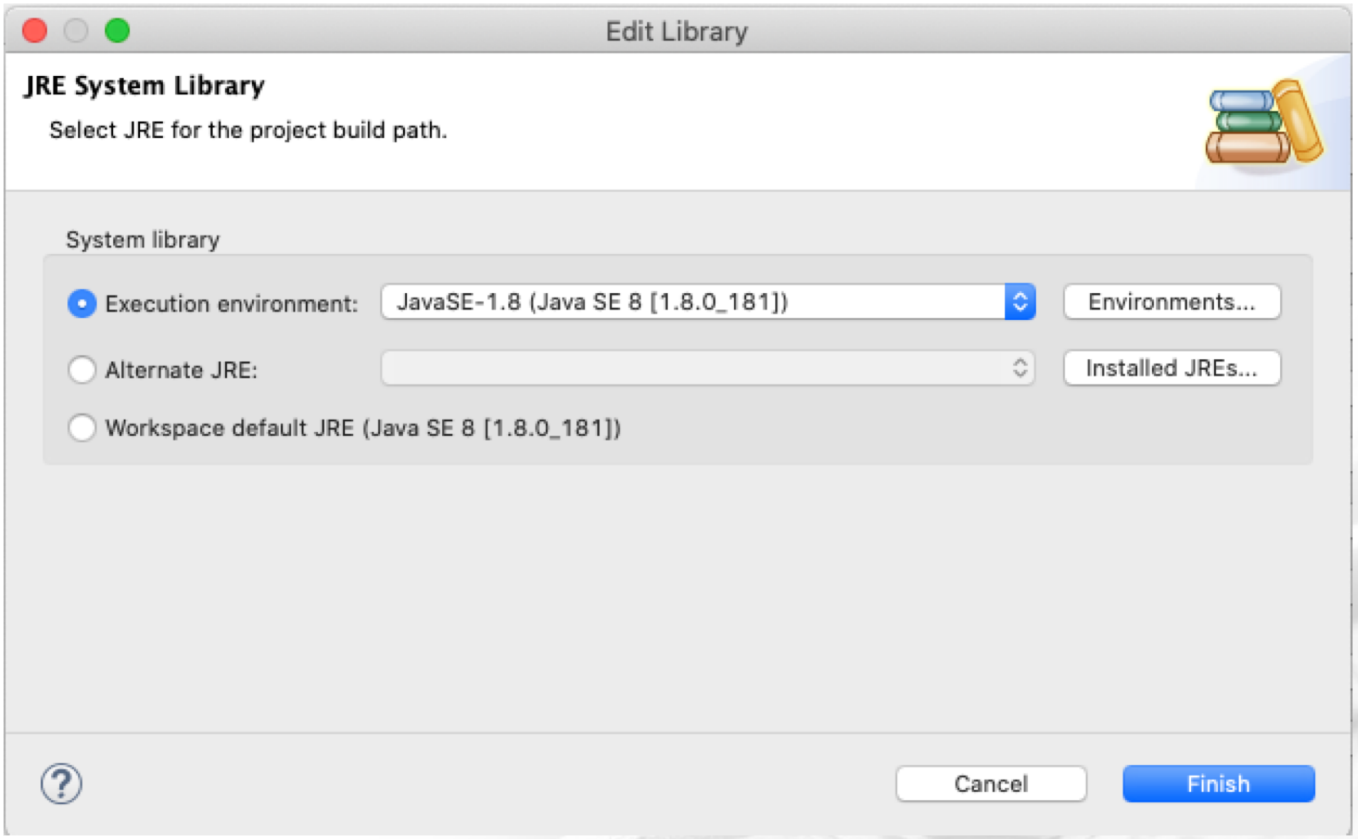
Spring

- Spring Java Application Project
  - ✓ Java 1.8 버전을 사용할 수 있도록 수정 – 프로젝트를 선택하고 마우스 오른쪽을 클릭하고 [properties]를 클릭하고 Build Path 수정



Spring

- Spring Java Application Project
  - ✓ Java 1.8 버전을 사용할 수 있도록 수정 – 프로젝트를 선택하고 마우스 오른쪽을 클릭하고 [properties]를 클릭하고 Build Path 수정



Spring

- Spring Java Application Project
  - ✓ 데이터를 표현하는 dto 클래스 생성(domain 패키지의 Item.java)
    - @Data를 했을 때 getter 와 setter 가 자동으로 만들어지지 않으면 getter&setter 그리고 toString을 추가

```
package domain;

import lombok.Data;

@Data
public class Item {
    private int itemId;
    private String itemName;
    private int price;
    private String description;
    private String pictureUrl;
}
```

Spring

- Spring Java Application Project
  - ✓ 데이터베이스에 접속해서 데이터를 읽어올 수 있는 클래스 생성 – persistence.ItemRepository

```
package persistence;

import domain.Item;

public class ItemRepository {
    public ItemRepository(){}

    public Item get() {
            Item item  = new Item();
            item.setItemId(1);
            item.setItemName("망고");
            item.setDescription("가장 좋아하는 과일");
            item.setPrice(3000);
            item.setPictureUrl("mango.jpg");
            return item;
    }
}
```

Spring

- Spring Java Application Project
  - ✓ 데이터베이스에 접속하는 메서드를 호출해서 출력하는 클래스 생성(Main.java)

```
import domain.Item;
import persistence.ItemRepository;

public class Main {
        public static void main(String[] args) {
                ItemRepository ItemRepository = new ItemRepository();
                Item item = ItemRepository.get();
                System.out.println(item);
        }
}
```

IoC

- 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ 인스턴스 생성에 정적 팩토리 메서드 사용
    - 장점
      - 다양한 방법으로 인스턴스를 생성하고자 하는 경우 생성자를 오버로딩 해서 구현할 수 있는데 생성자는 이름을 변경하지 못하기 때문에 직관성이 떨어짐
      - new를 이용하게 되면 인스턴스가 무조건 생성되는데 정적 팩토리 메서드를 이용하면 클래스 내부에 인스턴스를 생성한 후 반환하면 인스턴스를 여러 개 만들지 않을 수 있음
      - 생성자는 생성되는 객체의 클래스가 하나로 고정되지만 정적 팩토리 메서드를 이용하면 반환할 인스턴스의 클래스를 자유롭게 선택할 수 있음 – 매개변수에 따라 다른 클래스의 객체 반환 가능
      - 정적 팩토리 메서드를 만드는 시점에는 반환할 인스턴스의 클래스가 존재하지 않아도 됨
    - 단점
      - 생성자를 private 으로 만들면 상속 불가능
      - API 문서에서의 불편함

IoC

- ❖ 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ 인스턴스 생성에 정적 팩토리 메서드 사용
    - ● 정적 팩토리 메서드 네이밍
      - ◆ from: 매개변수를 하나 받아서 인스턴스를 반환
      - ◆ of: 여러 매개변수를 받아서 인스턴스를 반환
      - ◆ valueOf: from 과 of 의 자세한 버전
      - ◆ sharedInstance: 모두 동일한 인스턴스 반환
      - ◆ instance, getInstance: 매개변수를 받는다면 항상 동일한 인스턴스를 반환하지는 않음
      - ◆ create, newInstance: 항상 새로운 인스턴스를 생성해서 반환
      - ◆ getType: getInstance 와 동일하지만 현재 클래스가 아닌 다른 클래스의 인스턴스를 반환하는데 Type이 반환할 인스턴스의 타입
      - ◆ newType: newInstance 와 같으며 현재 클래스가 아닌 다른 클래스의 인스턴스를 반환
      - ◆ type: getType 과 newType의 간결화된 버전

IoC

- 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ 별도의 클래스에서 생성자를 호출해서 리턴하도록 구현하는 경우가 있음
    - 복잡한 인스턴스를 만들 때 조립하여 만들고자 할 때 – 빌더 패턴 고려
    - 특정 클래스에 의존하지 않도록 만들고 싶은 경우 - 모형이 되는 인스턴스를 등록해 두고 그 등록된 인스턴스를 복사해서 인스턴스를 생성
    - 인스턴스의 수명 주기 관리를 프레임워크에 위임하기 위해서

IoC

- 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ 프로젝트의 persistence 패키지에 RepositoryFactory 클래스 생성

```
package persistence;

public class RepositoryFactory {
    public static ItemRepository create() {
            return new ItemRepository();
    }
}
```

IoC

- 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ ItemRepository 클래스의 인스턴스를 다른 패키지에서 생성하지 못하도록 하기 위해서 생성자의 접근 지정자를 default 접근 지정자로 변경

```
~~public~~ ItemRepository(){ }
```

IoC

- 생성자 대신 정적 팩토리 메서드 또는 다른 클래스의 메서드를 호출해서 인스턴스를 생성
  - ✓ 프로젝트의 Main 클래스의 main 메서드 수정

```
import domain.Item;
import persistence.RepositoryFactory;
import persistence.ItemRepository;

public class Main {
    public static void main(String[] args) {
            ItemRepository repository = RepositoryFactory.create();
            Item item = persistence.get();
            System.out.println(item);
    }
}
```

IoC

- IoC(Inversion of Control – 제어의 역전 또는 역 흐름)
  - ✓ 개발자가 작성한 프로그램이 재사용 라이브러리의 흐름 제어를 받게 되는 소프트웨어 디자인 패턴
  - ✓ 전통적인 프로그래밍에서의 흐름은 프로그래머가 작성한 코드가 외부 라이브러리의 코드를 호출해서 사용
  - ✓ 제어의 역전이 적용된 구조에서는 외부 라이브러리의 코드가 프로그래머가 작성한 코드를 호출해서 사용
  - ✓ IoC의 목적
    - 작업을 구현하는 방식과 작업 수행 자체를 분리
    - 모듈을 제작할 때 모듈과 외부 프로그램의 결합에 대해 고민할 필요없이 모듈의 목적에 집중
    - 다른 시스템이 어떻게 동작할지에 대해 고민할 필요없이 미리 정해진 협약대로만 동작하게 하면 됨
    - 모듈을 바꾸어도 다른 시스템에 부작용을 일으키지 않음

IoC

- IoC
  - ✓ 스프링에서는 스프링이 제어권을 가지고 생성하고 관계를 설정하는 오브젝트를 Spring Bean이라고 함 – 안드로이드 나 React 에서는 Component 라고 함
  - ✓ Spring Bean은 스프링 컨테이너가 생성과 관계 설정, 사용 등을 제어해주는 제어의 역전이 적용된 오브젝트를 가리키는 단어
  - ✓ 스프링에서 Bean의 생성과 관계 설정 같은 제어를 담당하는 IoC 오브젝트가 Bean 팩토리(Bean Factory)
  - ✓ BeanFactory 인터페이스: bean 인스턴스를 관리하고 각 bean 인스턴스간의 의존 관계를 설정해주는 기능을 제공하는 가장 단순한 인터페이스

IoC

- Bean Factory
  - ✓ 스프링에서 다양한 부가 기능(웹 개발, 메시지 처리..)을 사용하기 위해서는 BeanFactory 인터페이스 대신에 여러 기능이 추가된 ApplicationContext 인터페이스를 이용
  - ✓ ApplicationContext
    - ApplicationContext는 IoC 컨테이너이면서 Singleton을 저장하고 관리하는 Singleton 레지스트리
    - 특별한 설정을 하지 않으면 스프링에서 생성한 Bean 오브젝트는 전부 Singleton 형태로 생성되고 매개변수가 없는 생성자(Default Constructor)를 이용해서 생성
    - Singleton으로 생성하는 이유는 스프링이 구동되는 환경이 서버 환경일 가능성이 높기 때문인데 서버 환경에서 클라이언트의 요청이 올 때마다 각 로직을 담당하는 오브젝트를 새로 만들게 되면 너무나 많은 오브젝트를 생성하게 되어서 성능이 저하될 가능성이 높아짐
    - 스프링의 Bean 오브젝트는 기본적으로 Singleton의 scope를 갖지만 요청이 올 때마다 생성해주는 prototype scope도 있고 웹에서 사용할 수 있는 request나 session scope도 존재
    - BeanFactory 인터페이스를 상속한 인터페이스로 Bean 인스턴스 라이프 사이클, 파일과 같은 자원 처리 추상화, 메시지 지원 및 국제화 지원, 이벤트 지원, xml 스키마 확장 등의 추가적인 기능을 제공하는 인터페이스
    - AnnotationConfigApplicationContext, GenericXmlApplicationContext 클래스가 ApplicationContext 인터페이스를 implements 한 클래스

IoC

- Bean Factory
  - ✓ WebApplicationContext
    - Web 프로젝트에서 사용가능 한 ApplicationContext 인터페이스
    - ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, XmlWebApplicationContext, AnnotationConfigWebApplicationContext 클래스가 implements
    - Spring Web Application에서는 설정 파일을 별도로 만들고 web.xml 파일에서 이 파일의 내용을 읽어서 XmlWebApplicationContext 인스턴스를 생성해서 사용

IoC

- Annotation을 이용한 ApplicationContext 이용
  - ✓ Factory 클래스 생성
    - 클래스 상단에 @Configuration을 추가
    - 인스턴스를 생성해서 리턴하는 메서드의 상단에 @Bean을 추가
  - ✓ Factory 클래스를 이용해서 Bean 생성
    - Configuration 클래스를 이용해서 AnnotationConfigApplicationContext 클래스의 인스턴스를 생성한 후 getBean을 호출해서 필요한 인스턴스를 생성
    - getBean 메서드에 문자열로 메서드의 이름만 대입해서 호출 할 수 있고 타입까지 포함해서 호출할 수 있는데 타입을 대입하지 않으면 Object 타입으로 리턴되므로 형변환해서 사용
    - 타입만 기재해서 호출하게 되면 동일한 자료형의 Bean이 있으면 그 Bean을 리턴

IoC

❖ Annotation을 이용한 ApplicationContext 활용 bean 생성

✓ 프로젝트의 RepositoryFactory 클래스 변경

```
package persistence;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RepositoryFactory {
        @Bean
        public static ItemRepository create() {
                return new ItemRepository();
        }
}
```

IoC

- Annotation을 이용한 ApplicationContext 활용 bean 생성
  - ✓ Main 클래스 변경

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import domain.Item;
import persistence.RepositoryFactory;
import persistence.ItemRepository;

public class Main {
        public static void main(String[] args) {
                //ItemRepository repository = RepositoryFactory.create();
                AnnotationConfigApplicationContext context =
                                new
    AnnotationConfigApplicationContext(RepositoryFactory.class);
                ItemRepository repository = context.getBean("create",
    ItemRepository.class);

                Item item = persistence.get();
                System.out.println(item);

                context.close();
        }
}
```

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ Spring Bean Configuration 파일을 만들고 <bean id="아이디" class="Bean의 클래스의 전체 경로" ></bean>을 이용해서 bean 생성 코드를 작성
  - ✓ xml 파일을 이용해서 GenericXmlApplicationContext 인스턴스를 생성하고 getBean 메서드 호출
  - ✓ xml을 이용한 Bean 오브젝트 생성을 위한 스프링 설정 파일

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="식별자" class="생성할 인스턴스의 경로">
    </bean>
</beans>
```

  - ✓ XML을 이용한 Bean 생성
    - @Configuration을 <beans>로 변경
    - @Bean을 <bean> 태그로 대응
    - 생성하는 메서드 이름을 bean 태그의 id 속성에 설정하고 클래스 이름을 class 속성에 설정

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ Spring Bean Configuration 파일 생성: [File] – [New] – [Spring] – [Spring Bean Configuration File] 을 선택해서 src/main/resources 디렉토리에 applicationContext.xml 파일을 생성하고 작성

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

    <bean id="ItemRepository" class="persistence.ItemRepository" />

</beans>
```

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ 설정한 정보를 가져와서 인스턴스를 생성하기 위해서 Main 클래스의 코드 수정

```
import org.springframework.context.support.GenericXmlApplicationContext;
import domain.Item;
import persistence.ItemRepository;
public class Main {
    public static void main(String[] args) {
            //ItemRepository repository = new ItemRepository();
            //ItemRepository repository = RepositoryFactory.create();
            //AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(RepositoryFactory.class);
            //ItemRepository repository = context.getBean("create",
    ItemRepository.class);
            GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("classpath:applicationContext.xml");
            ItemRepository repository = context.getBean("ItemRepository",
    ItemRepository.class);
            Item item = persistence.get();
            System.out.println(item);
            context.close();
    }
}
```

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ GenericXmlApplicationContext 클래스 대신 XmlBeanFactory(FileSystemResource인스턴스)를 이용해서 생성하는 방법도 있음
  - ✓ Spring Bean 설정 파일이 여러 개이면 파일 경로를 String [] 로 생성해서 GenericXMLApplicationContext 클래스의 생성자에 배열을 넘겨주어도 되고 경로가 같다면 *.xml을 이용해서 특정 디렉토리에 있는 모든 xml 파일을 설정 파일로 사용할 수 있음

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ Bean 설정 파일이 기본(src/main/resources) 디렉토리에 존재하는 경우에는 루트를 기준으로 기재 - db 패키지에 만든 경우

```
GenericXmlApplicationContext context =
new GenericXmlApplicationContext("classpath:/db/applicationContext.xml");
```

  - ✓ 파일 시스템에서 직접 참조하는 경우에는 classpath 대신에 file: 접두어를 이용

```
GenericXmlApplicationContext context =
new GenericXmlApplicationContext("file:src/main/java/db/applicationContext.xml " );
```

  - ✓ 특정 디렉토리에 있는 모든 XML 파일을 이용하고자 하는 경우에는 *을 이용

```
GenericXmlApplicationContext context =
new GenericXmlApplicationContext("classpath:/db/*.xml");
```

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ beans 태그
    - 스프링 설정 파일은 <beans> 태그를 루트 엘리먼트로 사용
    - beans 태그에 네임스페이스를 비롯한 XML 스키마 관련 정보를 설정
    - STS를 이용하여 만든 스프링 설정 파일에는 beans 네임스페이스가 기본 네임스페이스로 선언되어 있으면 sping-beans.xsd 스키마 문서가 schemaLocation으로 등록
    - 하위 태그로는 <bean>, <description>, <alias>, <import> 등이 있음
    - import 태그
      - 다른 설정 파일을 가져와서 사용할 때 사용하는 beans의 하위 태그
      - <import resource="다른 설정 파일 경로" />의 형식으로 사용

IoC

- xml을 이용한 ApplicationContext 이용
  - ✓ bean 태그
    - 클래스를 등록해서 인스턴스를 생성하기 위한 태그
    - 속성
      - class: 인스턴스를 생성할 클래스 경로로 필수
      - id: 외부에서 인스턴스를 사용하기 위해서 사용할 식별자로 설정하지 않으면 임의로 설정하는데 클래스 이름 뒤에 #0을 붙여서 설정
      - name: id에 대한 별명으로 특수문자 사용 가능
      - init-method: 초기화 메서드 이름 설정
      - destroy-method: 소멸될 때 호출되는 메서드 이름 설정
      - lazy-init: 지연 생성 여부로 true 또는 false 문자열로 설정
      - scope: 기본값은 singleton이고 호출할 때 마다 생성하는 prototype을 사용할 수 있음
      - abstarct: 다른 bean에 상속을 하기 위해서 생성하는 방식으로 외부에서 사용할 수는 없고 내부에서만 사용 가능한 설정으로 기본값은 false
      - parent: 상속받고자 하는 경우 상위 클래스의 bean 설정
      - factory-method: 생성자가 아닌 메서드(static 메서드)를 이용해서 생성하는 경우에 메서드 이름을 기재
      - depends-on: 다른 Bean에 의존해야 하는 경우 의존할 bean의 id 설정

DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI)
  - ✓ 프로그래밍에서 구성 요소간의 의존 관계를 하나의 인스턴스가 다른 인스턴스의 의존성을 제공하는 소프트웨어 디자인 패턴
  - ✓ 의존성: 인스턴스 내부에서 다른 클래스의 인스턴스를 사용하는 것
  - ✓ 주입: 사용하려는 인스턴스를 전달하는 것
  - ✓ 목적
    - 인스턴스의 생성 과 사용의 관심을 분리하는 것 – 결합도는 느슨하게 하고 의존 관계 역전 원칙 과 단일 책임 원칙을 따르도록 클라이언트의 생성에 대한 의존성을 클라이언트의 행위로부터 분리하는 것
    - 가독성이 높아짐
    - 코드 재사용 가능성이 높아짐
  - ✓ 의존성 주입은 광범위한 제어의 역전 테크닉의 한 형태
  - ✓ 어떤 서비스를 호출하려는 클라이언트는 그 서비스가 어떻게 구성되었는지 알 필요가 없도록 하고 클라이언트는 대신 서비스 제공에 대한 책임을 외부 코드로 위임하는데 클라이언트는 전달자 코드를 호출할 수 없도록 해야 하며 전달자는 이미 존재하거나 전달자에 의해 구성되었을 서비스를 클라이언트로 전달하고 클라이언트는 서비스를 사용

DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI)
  - ✓ 클라이언트가 전달자 와 서비스 구성 방식 또는 사용 중인 실제 서비스에 대해 알 필요가 없음을 의미하고 클라이언트는 서비스의 사용 방식을 정의하고 있는 서비스의 고유한 인터페이스에 대해서만 알면 됨
  - ✓ 구성 의 책임으로부터 사용 의 책임을 구분
  - ✓ 의존성 주입의 전제
    - 사용될 서비스 객체
    - 사용하는 서비스에 의존하는 클라이언트 객체
    - 클라이언트의 서비스 사용 방법을 정의하는 인터페이스
    - 서비스를 생성하고 클라이언트로 주입하는 책임을 갖는 주입자
  - ✓ 객체 지향 프로그래밍에서는 필요한 데이터를 개발자가 생성해서 사용하지만 스프링 프레임워크에서는 bean factory를 이용해서 생성한 후 주입 받아서 사용
  - ✓ 의존성 주입을 이용하는 방법은 생성자를 이용해서 주입받아 사용하거나 프로퍼티를 이용해서 주입받아 사용
  - ✓ 의존성 주입의 구현을 생성자를 이용하는 경우에는 클래스에 매개변수를 대입받는 생성자가 존재해야 하고 프로퍼티를 이용하는 경우에는 setter 메서드가 존재해야 함

DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI)
  - ✓ 생성자를 이용하는 경우

```
<bean id=" " class = " ">
  <constructor-arg  value = "값">
</bean>
```

  - ✓ 매개변수가 없는 생성자를 이용하는 경우에는 <constructor-arg> 생략
  - ✓ 매개변수가 있는 생성자를 이용하는 경우에는 bean 태그 안에 <constructor-arg> 태그의 속성으로 value 나 ref 에 전달하는 값 이나 bean 설정
  - ✓ constructor-arg 에서 타입을 설정하지 않으면 String 타입으로 처리하며 String이 아니면 가장 가까운 데이터 타입을 찾아서 주입
  - ✓ type 속성을 이용해서 직접 타입을 문자열 형식으로 기재 가능
  - ✓ value 대신 ref를 이용해서 다른 bean 태그의 id나 name을 사용하는 것이 가능한데 이 때는 ref bean="bean id" 으로 기재
  - ✓ 매개변수가 여러 개인 생성자를 호출하는 경우에는 constructor-arg 는 여러 개 사용이 가능한데 순서와 상관없이 일치하는 자료형을 찾아서 대입
  - ✓ 생성자의 매개변수가 여러 개 인 경우 index 속성을 이용해서 순서를 정해서 대입하는 것이 가능
  - ✓ 설정해야 하는 값이 null이면 여는 태그와 닫는 태그 사이에 <null />을 입력

DI(Dependency Injection)

- ❖ 의존성 주입(Dependency Injection, DI)
  - ✓ setter 메서드(property)를 이용하는 경우

```
<bean id=" " class = " ">
    <property name="setter 메서드에서 set을 제외한 부분">
        <value 또는 ref> 값 </value>
    </property>
</bean>
```

  - property name은 변수 이름이 아니고 setter 메서드에서 set을 제외한 부분의 첫 글자만 소문자로 변경한 문자열

DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI) – setter를 이용한 값 설정
  - ✓ applicationContext.xml 파일 수정

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

    <bean id="item" class="domain.Item">
            <property name="itemId"> <value>1</value> </property>
            <property name="itemName"> <value>키위</value> </property>
            <property name="price"> <value>2000</value> </property>
            <property name="description"> <value>비타민 C가 풍부한 과일</value>
    </property>
            <property name="pictureUrl"> <value>kiwi.jpg</value> </property>
    </bean>

</beans>
```

DI(Dependency Injection)

❖ 의존성 주입(Dependency Injection, DI) – setter를 이용한 값 설정

✓ Main 클래스를 수정

```
import org.springframework.context.support.GenericXmlApplicationContext;

import domain.Item;

public class Main {

    public static void main(String[] args) {
            try(GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("classpath:applicationContext.xml");) {
                    Item Item = context.getBean(Item.class);
                    System.out.println(Item);
            }
            catch(Exception e) {
                    e.printStackTrace();
            }
    }
}
```

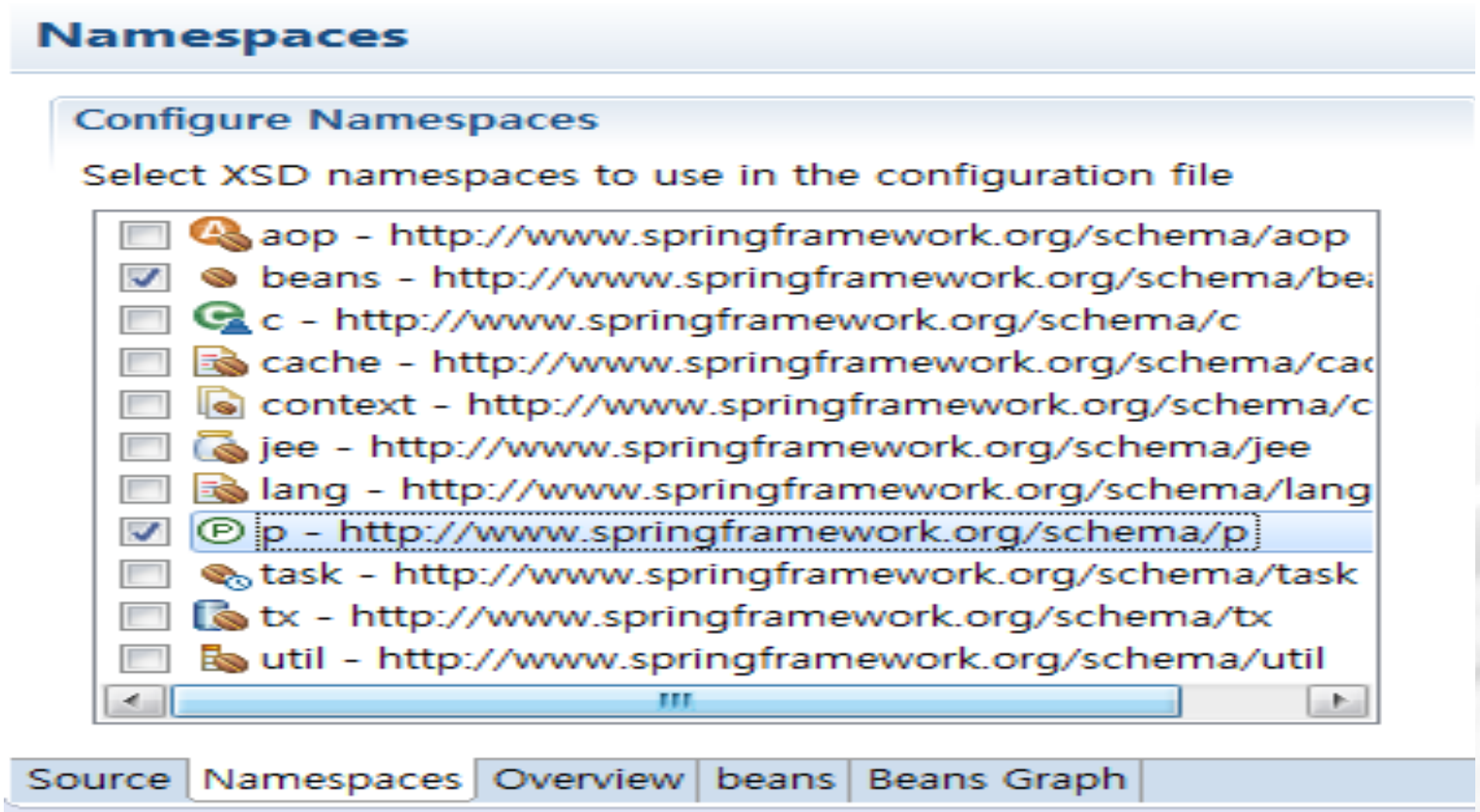
DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI) – setter를 이용한 값 설정
  - ✓ applicationContext.xml 파일 수정

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

    <bean id="itemId" class="java.lang.String">
            <constructor-arg> <value>2</value> </constructor-arg>
    </bean>

    <bean id="item" class="domain.Item">
            <property name="itemId"> <ref bean="itemId" /> </property>
            <property name="itemName"> <value>키위</value> </property>
            <property name="price"> <value>2000</value> </property>
            <property name="description"> <value>비타민 C가 풍부한 과일</value>
    </property>
            <property name="pictureUrl"> <value>kiwi.jpg</value> </property>
    </bean>

</beans>
```

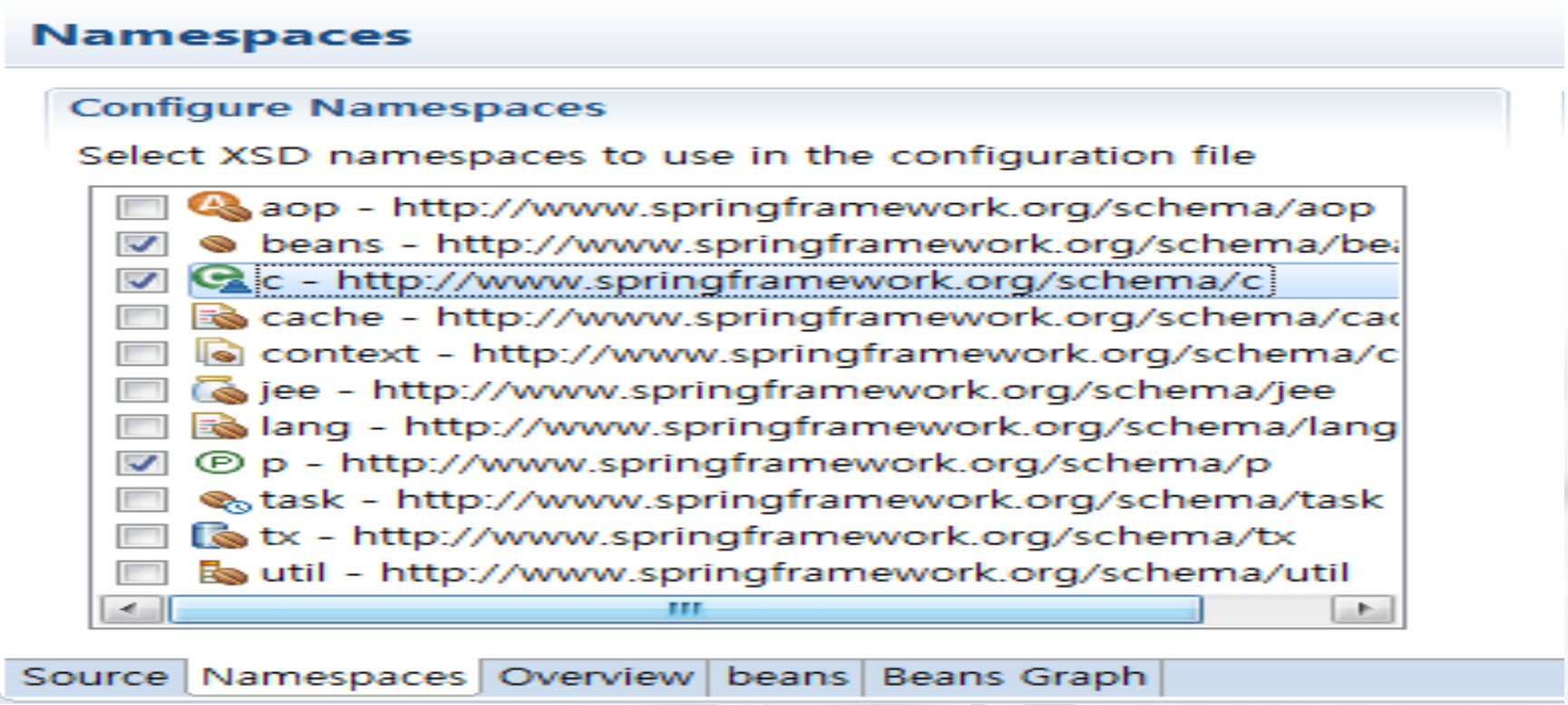
DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI)
  - ✓ **xmlns:p=http://www.springframework.org/schema/p** 네임스페이스를 이용하면 property name 대신에 bean 태그 안에 **p:이름** 을 이용해서 직접 값을 대입하는 것이 가능



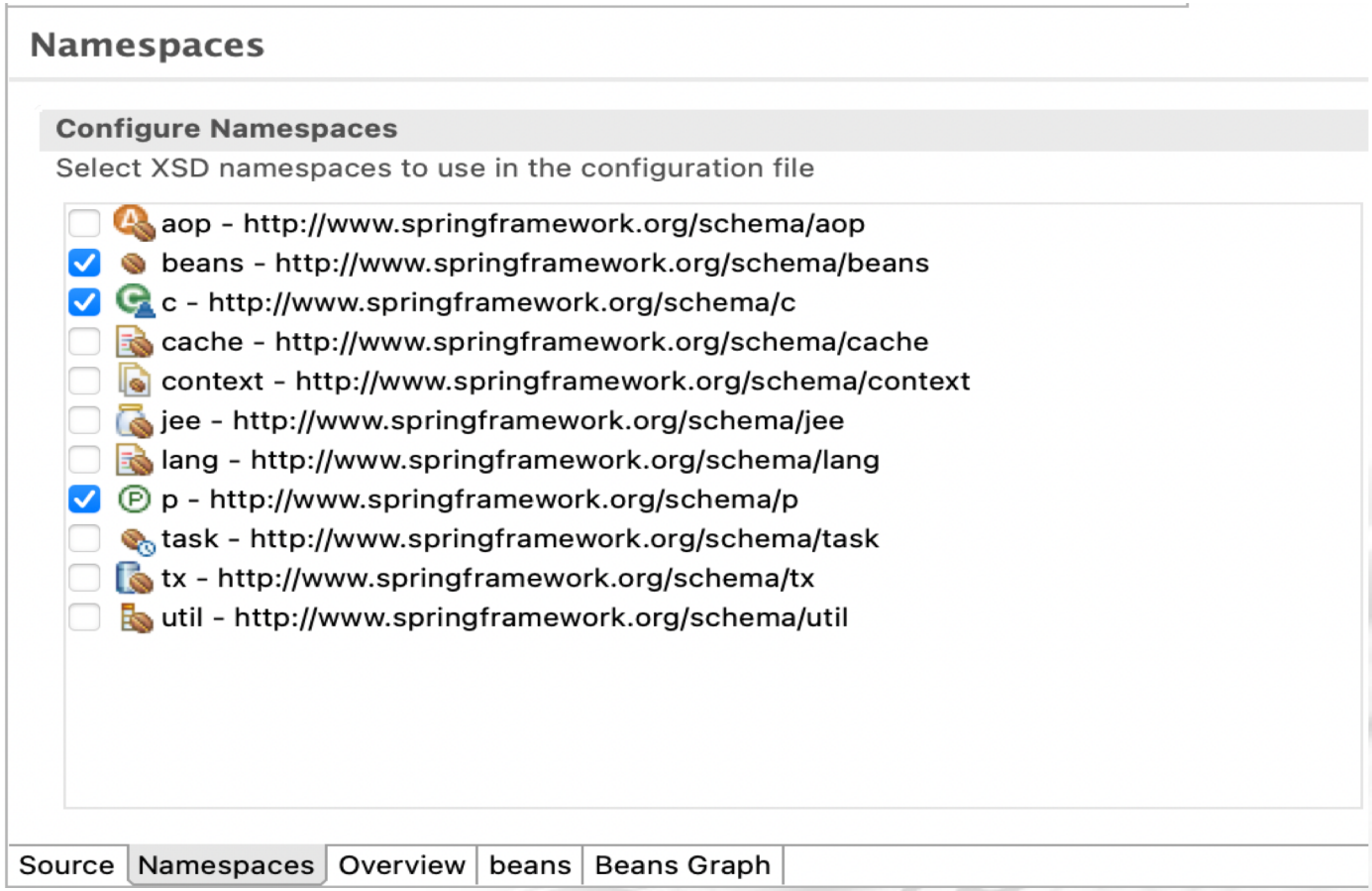
DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI)
  - ✓ **xmlns:c=http://www.springframework.org/schema/c** 의 네임스페이스를 이용하면 constructor-arg 대신에 **c:이름**을 이용해서 생성자에 직접 값을 대입하는 것이 가능
    - c:_인덱스를 이용해서 지정하는 것도 가능
    - 다른 bean을 참조할 때는 –ref를 추가하고 다른 Bean의 아이디를 추가



DI(Dependency Injection)

- 의존성 주입(Dependency Injection, DI) – p 와 c Namespace 이용
  - ✓ applicationContext.xml 파일의 내용 수정 – p 와 c namespace 추가 설정



DI(Dependency Injection)

❖ 의존성 주입(Dependency Injection, DI) – p 와 c Namespace 이용

✓ Item 클래스 수정

```
package domain;

import lombok.Data;

@Data
public class Item {
    private int itemId;
    private String itemName;
    private int price;
    private String description;
    private String pictureUrl;

    public Item() {}
    public Item(int itemId) {this.itemId = itemId;}
}
```

DI(Dependency Injection)

❖ 의존성 주입(Dependency Injection, DI) – p 와 c Namespace 이용

✓ applicationContext.xml 파일의 내용 수정

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="itemId" class="java.lang.String" >
            <constructor-arg> <value>3</value> </constructor-arg>
    </bean>
    <bean id="item" class="domain.Item" c:_0-ref = "itemId" p:itemId-ref="itemId"
    p:itemName="파인애플">
            <property name="price"> <value>2000</value> </property>
            <property name="description"> <value>비타민 C가 풍부한 과일</value>
    </property>
            <property name="pictureUrl"> <value>kiwi.jpg</value> </property>
    </bean>
</beans>
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ <list>: java.util.List 타입이나 자바의 배열의 데이터를 입력받기 위한 태그
  - ✓ <map>: java.util.Map 타입의 key와 value 목록을 입력받기 위한 태그
  - ✓ <set>: java.util.Set 타입의 데이터를 전달받기 위한 태그
  - ✓ <props>: java.util.Properties 타입의 데이터를 목록을 전달받기 위한 태그

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ List 타입

```
<list value-type="타입">
        <value>값</value>
        <value>값</value>
</list>
```

    - value의 기본 타입은 java.lang.String
    - 타입을 변경하고자 하면 <list value-type="실제타입">
    - 다른 bean 인스턴스를 참조하고자 하는 경우는 <ref bean="이름">
    - 비어 있는 인스턴스를 대입하고자 하는 경우는 <bean class="클래스이름">
  - ✓ Set 타입
    - 중복을 허용하지 않는 데이터의 집합으로 list 와 설정 방법 동일

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ Map

```
<map>
        <entry>
                <key><value>키값</value></key>
                <ref bean="아이디" />
        </entry>
        <entry key="키값" value="값" />
                <entry key="키값" >
                <value>값</value>
        </entry>
</map>
```

  - key, value의 기본 타입은 java.lang.String
  - 다른 bean 인스턴스를 참조하고자 하는 경우는 <ref bean="이름">
  - 비어 있는 인스턴스를 대입하고자 하는 경우는 <bean class="클래스이름">

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ Properties 타입: key와 value가 모두 String인 Map

```
<props>
    <prop key="키값">값</prop>
</props>
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ Bean으로 사용할 CollectionBean 클래스를 생성

```
package domain;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import lombok.Data;

@Data
public class CollectionBean {
    private List<String> list;
    private Set<String> set;
    private Map<String, Object> map;
    private Properties properties;
}
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ applicationContext.xml 파일에 bean 태그 추가

```
<bean id="collectionBean" class="domain.CollectionBean">

<property name="list">
        <list>
                <value>ArrayList</value>
                <value>LinkedList</value>
        </list>
</property>

<property name="set">
        <set>
                <value>HashSet</value>
                <value>LinkedHashSet</value>
                <value>TreeSet</value>
        </set>
</property>
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ applicationContext.xml 파일에 bean 태그 추가

```
<property name="map">
    <map>
        <entry>
            <key>
                <value>HashMap</value>
            </key>
            <value>키의 순서를 알 수 없음</value>
        </entry>
        <entry>
            <key>
            <value>LinkedHashMap</value>
            </key>
            <value>키는 저장한 순서대로</value>
        </entry>
        <entry>
            <key>
                <value>TreeMap</value>
            </key>
            <value>키의 크기 순서대로</value>
        </entry>
    </map>
</property>
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ applicationContext.xml 파일에 bean 태그 추가

```
        <property name="properties">
            <props>
                <prop key="IoC">인스턴스를 프레임워크가 생성하고 관리</prop>
                <prop key="DI">인스턴스 변수의 값을 외부에서 설정</prop>
            </props>
        </property>

    </bean>
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ Main 클래스 수정

```
public class Main {

    public static void main(String[] args) {
            try(GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("classpath:applicationContext.xml");) {
                    CollectionBean bean = context.getBean("collectionBean",
    CollectionBean.class);

                    System.out.println("==========List============");
                    for (String str : bean.getList()) {
                            System.out.println(str);
                    }

                    System.out.println("==========Set============");
                    for (String str : bean.getSet()) {
                            System.out.println(str);
                    }
```

DI(Dependency Injection)

- Collection Type을 주입받기 위한 스프링 태그
  - ✓ Main 클래스 수정

```
                        System.out.println("==========Map============");
                        Set<String> keySet = bean.getMap().keySet();
                        for (String key : keySet) {
                                System.out.println(key + ":" + bean.getMap().get(key));
                        }

                        System.out.println("==========Properties============");
                        Set<Object> propSet = bean.getProperties().keySet();
                        for (Object key : propSet) {
                                System.out.println(key + ":" +
        bean.getProperties().get(key));
                        }

                }
                catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - @Autowired 어노테이션을 이용하면 의존 관계를 자동으로 설정 가능
    - 변수와 메서드에 적용이 가능한데 변수에 적용하면 setter 메서드를 만들지 않아도 자동으로 setter 메서드를 만들어주고 동일한 자료형의 bean이 있으면 자동으로 연결
    - Autowired가 설정 된 경우 설정 파일에 동일한 자료형으로 만들어진 bean이 없다면 아래와 같은 예외가 발생

```
Exception in thread "main"
    org.springframework.beans.factory.BeanCreationException: Error creating bean
    with name 'memberRepository': Injection of autowired dependencies failed;
    nested exception is org.springframework.beans.factory.BeanCreationException:
    Could not autowire field: di.Member di.MemberRepository.data; nested
    exception is
    org.springframework.beans.factory.NoSuchBeanDefinitionException: No
    qualifying bean of type [di.Member] found for dependency: expected at least
    1 bean which qualifies as autowire candidate for this dependency.
    Dependency annotations:
    {@org.springframework.beans.factory.annotation.Autowired(required=true)}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - @Autowired가 설정되면 반드시 데이터를 생성해서 대입해주어야 하는데 필수 요소 지정 해제는 @Autowired(required=false)를 설정
    - Autowired가 설정 된 경우 설정 파일에 동일한 자료형으로 만들어진 bean이 2개 이상 존재한다면 예외가 발생
    - 동일한 자료형으로 만들어진 Bean이 2개 이상이면 @Autowired 하단에 @Qualifier("Bean의 아이디 ")를 설정해서 특정한 Bean을 연결하도록 설정
    - 타입 -> 이름 -> @Qualifier -> 실패 순으로 동작
    - Java 1.8 버전을 사용하는 프로젝트에서는 @Autowired 어노테이션 대신에 @Resource(name="이름") 어노테이션을 사용할 수 있는데 이 경우에는 자료형이 아니라 이름을 가지고 매핑하는데 이름 -> 타입 -> @Qualifier -> 실패 순으로 동작
    - @Inject 와 @Named 도 존재하는데 이 경우는 javax.inject 라이브러리의 의존성을 추가해야 하며 타입 -> @Qualifier -> 이름 -> 실패 순으로 동작

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - domain.User 클래스 생성 - @Data를 했을 때 getter 와 setter 가 자동으로 만들어지지 않으면 getter&setter 그리고 toString을 추가해주면 됩니다.

```
package domain;

import lombok.Data;

@Data
public class User {
    private String id;
    private String pw;
}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - UserRepository 클래스 생성

```
package persistence;

import domain.User;

public class UserRepository {
    public User getUser() {
        User user = new User();
        user.setId("itstudy");
        user.setPw("1234");
        return user;
    }
}
```

DI(Dependency Injection)

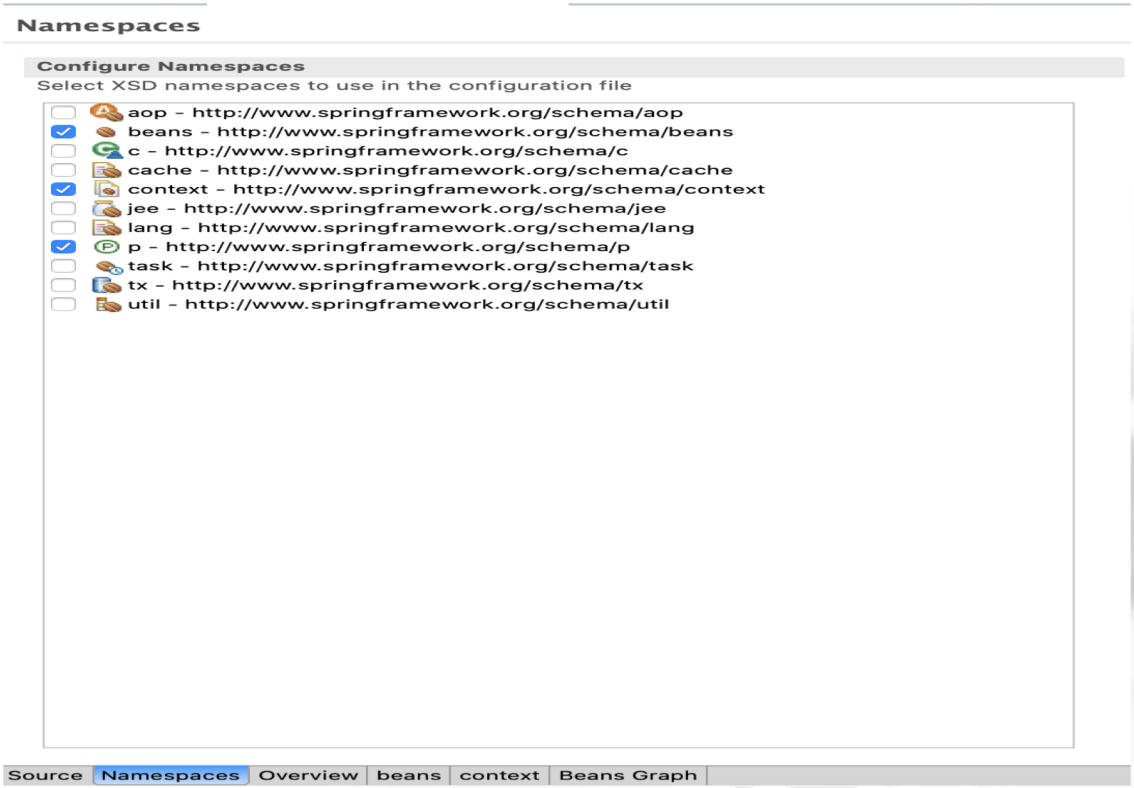
- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - UserService 클래스 생성

```
package service;

import org.springframework.beans.factory.annotation.Autowired;

import persistence.UserRepository;
import domain.User;

public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User getUser(){
       return userRepository.getUser();
    }
}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ Annotation 과 XML 설정 파일 같이 사용
    - context 네임 스페이스를 추가
    - 설정 파일에 추가

```
<context:annotation-config />
```



DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - applicationContext.xml 파일에 코드 추가

```
<context:annotation-config />

<bean id="userService" class="service.UserService" />
<bean id="userRepository" class="persistence.UserRepository" />
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 - @Autowired
    - main 메서드에 작성하고 실행

```
import org.springframework.context.support.GenericXmlApplicationContext;

import anno.service.UserService;

public class Main {
    public static void main(String[] args) {
      try (GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("classpath:applicationContext.xml");) {
                UserService service = context.getBean(UserService.class);
                System.out.println(service.getUser());
      } catch (Exception e) {
                e.printStackTrace();
      }
    }
}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 의존성 자동 주입 – 설정 파일 활용
    - bean의 autowire 속성에 byType을 설정하면 인스턴스 속성 과 동일한 자료형의 bean이 있으면 자동 주입 받을 수 있음
    - autowire 속성을 byName 속성을 설정하면 동일한 id를 사용하는 bean이 있으면 자동 주입
    - autowire 속성을 default로 설정하거나 no로 설정하면 자동 주입을 하지 않음
    - autowire-candidate 속성을 false로 설정하면 다른 bean 이 이 bean을 자동 주입할 수 없도록 함

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 생성자를 이용한 의존성 자동 주입 – @ConstrutorProperties
    - UserService 클래스 수정

```
@Service
public class UserService {

    private final UserRepository userRepository;

    @ConstructorProperties({"userRepository"})
    public UserService(UserRepository userRepository) {
       this.userRepository = userRepository;
    }

    public User getUser(){
       return userRepository.getUser();
    }
}
```

DI(Dependency Injection)

- annotation을 이용한 DI
  - ✓ 생성자를 이용한 의존성 자동 주입 – RequiredArgsConstructor 어노테이션 이용
    - UserService 클래스 수정

```
package service;

import persistence.UserRepository;
import domain.User;
import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;

    public User getUser(){
       return userRepository.getUser();
    }
}
```

DI(Dependency Injection)

- Bean 자동 생성
  - ✓ 프로젝트에서 모든 클래스가 특정 패키지에 모두 속해있다면 클래스의 bean을 직접 생성하는 것은 코드의 낭비
  - ✓ Component Scan을 이용하면 특정한 패키지에 속한 Bean을 자동으로 생성
  - ✓ 자동으로 생성할 Bean 클래스의 상단에 @Component를 추가하고 설정 파일에서 <context:component-scan base-package="패키지이름"></context:component-scan> 을 설정하면 패키지이름에 속한 클래스 중에서 @Component 어노테이션이 적용된 클래스의 인스턴스를 자동으로 bean으로 등록해서 생성
  - ✓ Annotation을 이용해서 scan을 하고자 할 때는 @ComponentScan(basePackages={"패키지이름"}) 을 이용
  - ✓ bean의 이름은 자동으로 클래스 이름의 첫글자만 소문자로 변경해서 설정하는데 이름을 변경하고자 하면 @Component("변경할 이름")으로 설정
  - ✓ @Component 와 동일한 효과를 나타내주는 @Controller, @Service, @Repository 어노테이션도 있음

DI(Dependency Injection)

- Bean 자동 생성
  - ✓ UserRepository 클래스의 상단에 어노테이션 추가

```
@Repository
public class UserRepository {
    public User getUser(){
            User user = new User();
            user.setId("ggangpae");
            user.setPw("1234");
            return user;
    }
}
```

DI(Dependency Injection)

- Bean 자동 생성
  - ✓ UserService 클래스의 상단에 어노테이션 추가

```
@RequiredArgsConstructor
@Service
public class UserService {
    private final UserRepository userRepository;

    public User getUser(){
            return userRepository.getUser();
    }
}
```

DI(Dependency Injection)

- Bean 자동 생성
  - ✓ applicationContext.xml 파일에 컴포넌트 스캔 코드를 추가하고 실행

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:c="http://www.springframework.org/schema/c"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:annotation-config />
    <!--
    <bean id="userService" class="service.UserService" />
    <bean id="userRepository" class="persistence.UserRepository" />
     -->
    <context:component-scan base-package="service, persistence" />

</beans>
```

DI(Dependency Injection)

- Bean 자동 생성
  - ✓ RepositoryFactory 클래스에서 에러가 발생하면 상단의 어노테이션 주석 제거

```
//@Configuration
public class RepositoryFactory {
    @Bean
    public static ItemRepository create() {
            return new ItemRepository();
    }
}
```

DI(Dependency Injection)

- bean 태그의 속성 관련 어노테이션
  - ✓ @Scope
    - 스코프를 지정하는 기본은 Singleton: @Scope(value=ConfigurableBeanFactory.SC0PE_PR0T0TYPE) 의 형태로 지정
  - ✓ @Lazy
    - 지연 생성 여부를 설정: @Lazy(value=true) 의 형태로 지정
  - ✓ @DependsOn
    - 의존 관계를 설정하는 것으로 특정 Bean의 생성을 먼저 하도록 설정: @DependsOn(value={"Repository"} 의 형태로 지정
  - ✓ @Primary
    - 동일한 자료형의 bean 이 여러 개 있을 때 우선해서 의존 관계 설정에 사용하도록 하는 것

DI(Dependency Injection)

- 스프링 컨테이너의 수명 주기
  - ✓ 컨테이너 생성
  - ✓ Bean 메타 정보를 이용해서 Bean 인스턴스 생성
  - ✓ 컨테이너 사용
  - ✓ 컨테이너 종료(Bean 인스턴스 제거)
    - ApplicationContext 인스턴스를 생성하면 1번과 2번이 수행
    - getBean 등의 메서드를 호출하는 것이 3번
    - close()를 호출하면 Spring Bean이 전부 소멸
    - 자바 프로세스를 강제로 종료(Ctrl+C)하는 경우에는 close()가 호출되지 않을 수 있는데 이 경우에는 registerShutdownHook()을 호출하도록 하면 자바 프로세스를 강제로 종료하는 경우에도 스프링 컨테이너를 종료할 수 있으며 운영체제 차원에서 종료하는 경우에는 역시 호출되지 않음

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ Spring Bean 인스턴스의 초기화와 소멸 시 특정 메서드를 호출해야 하는 경우에는 리턴 타입이 void 이고 매개변수가 없는 메서드를 Bean의 클래스에 생성하고 상단에 @PostConstruct 와 @PreDestroy 어노테이션을 추가
    - pom.xml 파일의 dependencies 에 추가

```
<dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>javax.annotation-api</artifactId>
        <version>1.3.2</version>
</dependency>
```

DI(Dependency Injection)

- 스프링 컨테이너의 수명 주기
  - ✓ Spring Bean 인스턴스의 초기화와 소멸 시 특정 메서드를 호출해야 하는 경우에는 리턴 타입이 void 이고 매개변수가 없는 메서드를 Bean의 클래스에 생성하고 상단에 @PostConstruct 와 @PreDestroy 어노테이션을 추가
    - UserRepository 클래스에 메서드 추가

```
@PostConstruct
public void annoInit(){
    System.out.println("어노테이션을 이용한 초기화 메서드");
}

@PreDestroy
public void annoDestroy(){
    System.out.println("어노테이션을 이용한 소멸자 메서드");
}
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ 어노테이션 대신에 InitializingBean, DisposableBean 인터페이스를 implements 해서 작성 가능
    - UserRepository 클래스 수정

```
@Repository
public class UserRepository implements InitializingBean, DisposableBean {
    public User getUser() {
        User user = new User();
        user.setId("itstudy");
        user.setPw("1234");
        return user;
    }

    @PostConstruct
    public void annoInit(){
        System.out.println("어노테이션을 이용한 초기화 메서드");
    }

    @PreDestroy
    public void annoDestroy(){
        System.out.println("어노테이션을 이용한 소멸자 메서드");
    }
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ 어노테이션 대신에 InitializingBean, DisposableBean 인터페이스를 implements 해서 작성 가능
    - UserRepository 클래스 수정

```
    @Override
    public void destroy() throws Exception {
       System.out.println("인터페이스를 이용한 소멸자 메서드");
    }
    @Override
    public void afterPropertiesSet() throws Exception {
       System.out.println("인터페이스를 이용한 초기화 메서드");
    }

}
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ Spring Bean 인스턴스의 초기화와 소멸 시 bean태그의 init-method 속성과 destroy-method 속성에 메서드 이름을 설정하는 방법도 제공하는데 이것이 custom init과 커스텀 custom destroy
    - domain.User 클래스 수정

```
@Data
public class User {
    private String id;
    private String pw;

    public void init() {
       System.out.println("User 생성");
    }
    public void exit() {
       System.out.println("User 소멸");
    }
}
```

DI(Dependency Injection)

- 스프링 컨테이너의 수명 주기
  - ✓ Spring Bean 인스턴스의 초기화와 소멸 시 bean태그의 init-method 속성과 destroy-method 속성에 메서드 이름을 설정하는 방법도 제공하는데 이것이 custom init과 커스텀 custom destroy
    - applicationContext.xml 파일에 추가

```
<bean class="domain.User" id="user" init-method="init" destroy-method="exit">
   <property name="id"> <value>ggangpae1</value> </property>
   <property name="pw"> <value>1234</value> </property>
</bean>
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ BeanPostProcessor
    - Bean 객체를 정의할 때 init-method 속성을 설정하면 객체가 생성될 때 자동으로 호출될 메서드를 지정할 수 있는데 이 때 BeanPostProcessor 인터페이스를 구현한 클래스를 정의하면 Bean 객체를 생성할 때 호출될 init 메서드 호출을 가로채 다른 메서드를 호출할 수 있도록 할 수 있음
    - 메서드
      - ◆ postProcessBeforInitialization: init-method에 지정된 메서드가 호출되기 전에 호출
      - ◆ postProcessAfterInitialization: init-method에 지정된 메서드가 호출된 후에 호출
    - init-method가 지정되어 있지 않더라도 자동으로 호출됨

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ BeanPostProcessor
    - ● 프로젝트에 클래스 추가 – util.TestBeanPostProcessor

```
package util;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
public class TestBeanPostProcessor implements BeanPostProcessor{
	// init-method 호출 전
	public Object postProcessBeforeInitialization(Object bean, String beanName)
					throws BeansException {
		System.out.println("before");
		System.out.println(beanName);
		return bean;
	}

	// init-method 호출 후
	public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
		System.out.println("after");
		return bean;
	}
}
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ BeanPostProcessor
    - ● applicationContext.xml 파일에 bean 생성 코드 추가

```
<bean class="util.TestBeanPostProcessor" />
```

DI(Dependency Injection)

- ❖ 스프링 컨테이너의 수명 주기
  - ✓ DestructionAwareBeanPostProcessor
    - Bean 객체가 소멸될 때 상호 작용을 하기 위한 인터페이스
    - 메서드
      - ◆ postProcessBeforeDestruction(Object bean, String beanName): 소멸될 때 호출되는 메서드

DI(Dependency Injection)

- Bean의 라이프 사이클
  1. Bean 인스턴스 생성
  2. Bean 프로퍼티 설정
  3. BeanNameAware의 setBeanName()
  4. ApplicationContextAware의 setApplicationContext()
  5. @PostConstruct 메서드
  6. InitializingBean 인터페이스 메서드
  7. 커스텀 init
  8. Bean 인스턴스 사용
  9. @PreDestroy 메서드
  10. DisposableBean 인터페이스의 메서드
  11. 커스텀 destroy 메서드
  12. Bean 인스턴스 소멸

DI(Dependency Injection)

- 스프링 컨테이너 접근
  - ✓ Bean 인스턴스에서 스프링 컨테이너에 접근해야 하는 경우가 있을 수 있는데 이 때는 ApplicationContextAware 인터페이스를 이용할 수 있고 Bean의 이름을 사용하고 싶다면 BeanNameAware 인터페이스를 이용
  - ✓ ApplicationContextAware 인터페이스는 Bean의 초기화 과정에서 ApplicationContext를 전달
  - ✓ BeanNameAware 인터페이스는 초기화 과정에서 Bean 이름을 전달

```
public class UserRepository implements ApplicationContextAware, BeanNameAware {

    private ApplicationContext context;
    private String beanName;

    @Override
    public void setBeanName(String arg0) {
            beanName = arg0;
    }

    @Override
    public void setApplicationContext(ApplicationContext arg0) throws BeansException {
            context = arg0;
    }
}
```

DI(Dependency Injection)

- bean의 scope
  - ✓ 스프링의 bean은 기본적으로 singleton으로 생성되며 이 singleton은 클래스 차원이 아니라 컨테이너 차원에서의 singleton을 의미
  - ✓ bean 태그 내의 scope 속성을 이용해서 범위를 설정
  - ✓ singleton, prototype(사용 시 마다 생성), request, session 등을 지정
  - ✓ scope는 절대적인 개념이 아니어서 만일 singleton 범위의 bean이 prototype 범위의 bean을 참조한다면 이 경우에는 참조되는 bean의 scope가 singleton으로 변경됨
  - ✓ annotation을 이용해서 범위를 적용할 때는 @Bean 아래에 @Scope("범위")를 설정

JUnit Test

- Test
  - ✓ 스프링 프로젝트에서는 Spring-Test 와 Junit 라이브러리를 이용해서 테스트하는 것이 일반적
  - ✓ JUnit 라이브러리는 테스트를 위한 라이브러리로 메서드 위에 @Test라는 어노테이션을 사용하면 프로그램 전체를 완성하지 않아도 실행 가능
  - ✓ 스프링에서 Test 클래스를 만드는 방법은 src/test/java 디렉토리에 클래스를 만들고 클래스 이름 위에 @RunWith(SpringJUnit4ClassRunner.class)를 추가
  - ✓ 스프링 설정 파일의 내용을 실행하고자 할 때는 @ContextConfiguration("설정파일 경로")

JUnit Test

- ✓ pom.xml 파일에서 Junit 라이브러리의 의존성 확인

```
<!-- Test -->
<junit.version>4.12</junit.version>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

JUnit Test

- pom.xml 파일에 spring-test 의존성이 설정되어 있는지 확인하고 scope 삭제

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring-framework.version}</version>
    <scope>test</scope>
</dependency>
```

JUnit Test

- src/test/java 디렉토리에 테스트용 클래스를 만들고 테스트

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;

import service.UserService;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SampleTesting {
    @Autowired
    private UserService userService;

    @Test
    public void method() {
       System.out.println(userService.getUser());
    }
}
```