Spring Framework

Spring 4.1

대덕인재개발원 : 최희연

❑ 이 교재는

- 이 교재는 전자정부 표준 프레임위크 의 "실행 환경 교재"를 기반으로 작성되었으며 Spring 4.1에 맞추어 변경/추가 되었습니다.

❑ 참고자료

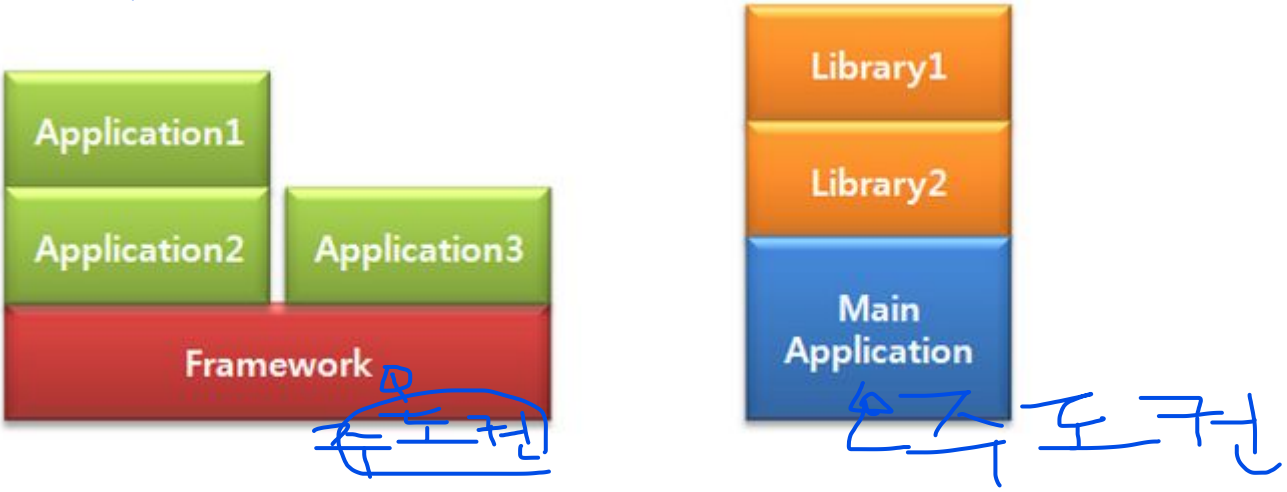
- 스프링 레퍼런스
  - http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/
- 전자정부 표준 프레임위크 사이트 / 실행 환경 교재
  - http://www.egovframe.go.kr/wiki/doku.php
- 토비의 스프링 3 : 스프링 프레임워크 3 기초 원리부터 고급 실전활용까지 완벽 가이드
  - 저자 : 이일민 , 출판사 : 에이콘, 출판년도 : 2010/08/05
- 웹 개발자를 위한 스프링 4.0 프로그래밍
  - 저자 : 최범균, 출판사 : 가메출판사, 출판년도 : 2014/08/15
- Toby's Epril 블러그(이일민)
  - http://toby.epril.com/
- Anyframe
  - http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/htmlsingle/core.html

❑ 참고

- Framwork 와 Library 의 차이
  - Framework : 소프트웨어의 특정 문제를 해결하기 위해 상호협력하는 클래스와 인터페이스의 집합.
    1. 특정 개념들의 추상화를 제공하는 클래스나 컴포넌트들로 구성
    2. 상기의 추상적 개념들이 문제를 해결하기 위해 협업하는 방법을 정의함.
    3. 재사용 가능한 컴포넌트들의 집합
    4. 보다 수준 높은 패턴들로 조직화된 구조.
  - Library : 소프트웨어에서 호출할수 있는 함수와 루틴들로 구성

IoC Inversion of Control

↳ P.10 ⇒ DI



❑ History

- 2002,3년에 Rod Johnson 과 Juergen Holler에 의해서 시작되었으며
- Rod Johnson의 저서 "Expert One-on-One J2EE Design and Development" 출판과 함께 프레임워크로 개발 시작됨 (2004년도에 "Expert One-on-One J2EE Development without EJB" 로 새롭게 출판)
- 2004년 3월 Spring 1.0이 릴리즈됨

❑ EJB 명세와 현실의 괴리

- 원격 호출 기반의 EJB는 객체에 대해서 무거운(heavy weight)모델임
- 대부분의 개발자들은 스테이트리스 세션 빈과 비동기 방식이 필요한 경우에 한해서 메시지 드리븐 빈만을 사용함

❑ EJB 의 실패 부분

- 너무 복잡함
- 저장을 위한 엔티티 빈은 실패작임
- EJB의 이식성은 서블릿과 같은 다른 J2EE 기술보다 떨어짐
- 확장성을 보장한다는 EJB의 약속과 달리, 성능이 떨어지며 확장이 어려움

❑ Spring Framework

- Spring이라는 이름의 기원은 전통적인 J2EE를 "겨울"에 빗대어 "겨울" 후의 "봄"으로 새로운 시작을 의미함
- EJB가 제공했던 대부분의 기능을 일반 POJO(Plain Old Java Object)를 사용하여 개발할 수 있도록 지원함
- 엔터프라이즈 어플리케이션 개발의 복잡성을 줄이기 위한 목적으로 개발됨

❑ Introduction

- JavaEE 기반의 어플리케이션 개발을 쉽게 해주는 오픈 소스 어플리케이션 프레임워크
- 전체 애플리케이션을 체계적으로 엮어낼(wire up) 수 있는 프레임워크
- POJO기반의 개발을 통해 외곽적인 코드 없이 빈들에 대한 생명주기를 관리
- 트랜잭션 관리를 위한 일관된 방법을 제공
- O/R mapping : Hibernate, iBatis, JDO등과의 연동 시의 필요 작업 최소화
- Lightweight Container 로 시작 시 부하가 적음
- 다양한 3rd 파티 제품과의 연동
- 테스트의 용이성

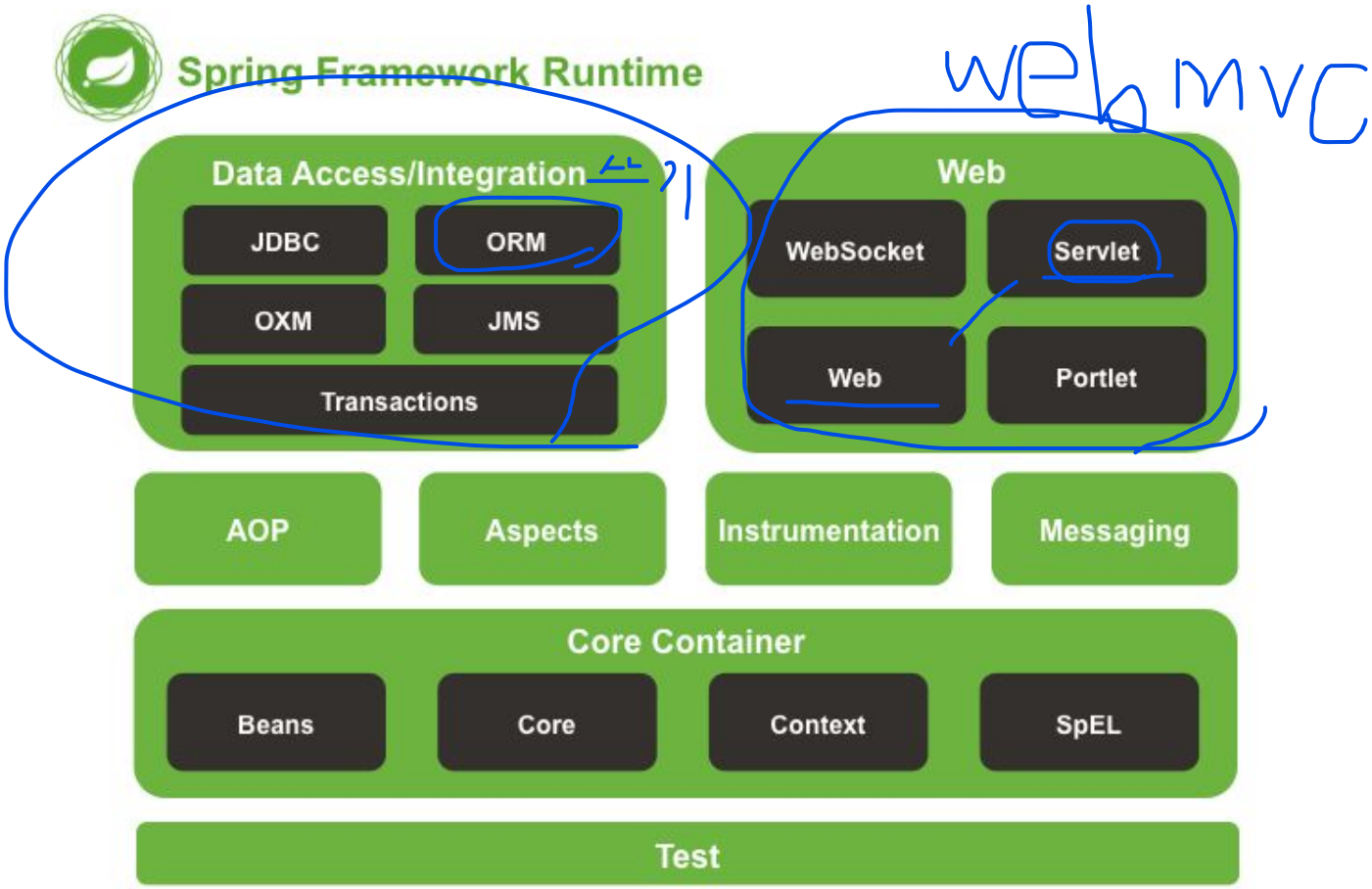
❑ Spring 의 기본 전략

- POJO 를 이용한 가볍고 가능한 비침투적인(non-invasive)개발
- DI 와 인터페이스 지향을 통한 느슨한 결합도(loose coupling)
- Aspect 와 공통 규약을 통한 선언적 프로그래밍
- Aspect 와 템플릿을 통한 상투적인 코드의 축소.

❑ Spring 포트폴리오

- Spring web flow : MVC 프레임워크를 기반으로 목표를 향해 사용자를 안내하는 대화형, 흐름 기반 웹 어플리케이션의 구축을 지원.
- Spring web service : contract-first 웹서비스 모델을 제공, WSDL을 먼저 작성 후 비즈니스 로직을 구현하는 방식.
- Spring Security : 필터와 AOP 를 기반으로 구현된 선언적 보안 메커니즘 제공 프레임워크.
- Spring Integration : 엔터프라이즈 어플리케이션들간의 통합 패턴의 구현체를 제공.
- Spring Batch : 데이터에 대한 일괄처리 작업을 지원해주는 프레임워크.
- Spring Social : SNS 와의 연동을 지원해주는 프레임워크
- Spring mobile & Spring Android : 모바일 웹 어플리케이션의 개발을 지원하는 확장 모듈.
- Spring DM : OSGi 프레임워크와 Spring 의 선언적 DI 방식을 엮어서 OSGi 내에서 서비스를 선언적으로 발행하고, 소비하는 HCLC(High Cohesion, Loose Coupling) 어플리케이션의 개발을 지원.
- Spring LDAP : LDAP 에 스프링의 템플릿 기반 엑세스를 제공하는 프레임워크.
- Spring rich client : Swing 어플리케이션 개발을 지원해주는 모듈.
- Spring .NET : .NET 플랫폼에서 스프링 DI 와 AOP 를 지원해주는 프레임워크.
- Spring Flex : 강력한 RIA 플랫폼인 Flex, AIR 와의 연동으로, BlazeDS를 통한 서버측 스프링 빈 이용이 가능, flex 어플리케이션의 rapid development 을 지원하는 프레임워크.
- Spring Roo : 자바 어플리케이션의 신속 개발을 지원하는 일종의 개발 환경.
- 기타 Spring extensions(Spring data, python 용 스프링 구현체등...)

❑ Spring Overview



❑ Core Container

- Core and Beans
  - Spring 프레임워크의 근간이 되는 IoC/DI 기능을 지원하는 영역을 담당하고 있다.
  - BeanFactory를 기반으로 Bean 클래스들을 제어할 수 있는 기능을 지원한다.
- Context
  - Core and Beans를 견고히(solid)한 모듈이다. JNDI 처럼, 프레임워크 방식으로 객체에 접근하는 방법을 제공
  - Beans 모듈에 더하여 국제화, 이벤트 전파, 리소스 로딩, 투명한 context 생성 등을 제공한다.
- Expression Language(SpEL)
  - 객체 탐색을 실행 시에 구하거나(querying) 조작할 수 있도록 강력한 표현언어를 제공한다.(SpringEL/SpEL)
  - 객체 속성에 대한 값 읽기 및 설정, 메서드 호출, 배열에 접근, collection and indexers, 논리 및 수리 연산, named variables, Spring의 IoC container로부터 이름에 의한 객체 검색 등을 제공한다.

❑ AOP and Instrumentation

- AOP(Aspect-oriented programming)
  - AOP Alliance 기반의 Aspect Oriented Programming을 지원한다.
  - 업무로직에서 부가적인 기능들을 method-interceptors, pointcuts 을 이용하여 분리(module)하여 작성할 수 있다.
- Instrumentation
  - 어플리케이션 서버에서 사용되도록 클래스 구현의 지원이나 클래스 로더의 구현을 제공한다.
  - 단어 : instrument (네이버)
    장치나 도구를 말하며, 기능을 확대하기 위한 보조 장치나 보조 기구를 뜻하는데 측정 기기, 사진 기기 등이 포함된다.

❑ Data Access/Integration

- JDBC
  - JDBC 기반하의 DAO개발을 좀 더 쉽고, 일관된 방법으로 개발하는 것이 가능하도록 추상화된 레이어를 제공.
- ORM
  - Object Relation Mapping 프레임워크인 Hibernate, iBatis, JDO, JPA와의 통합을 지원.
- OXM
  - Object/XML Mapping 은 Object와 XML간의 변환을 위한 추상 계층을 제공한다. (JAXB, Castor, XMLBeans, JiBX, XStream 등)
- JMS
  - Java Message Service, 메시징 처리를 위한 모듈을 제공한다.
- Transaction
  - 직접적인 트랜잭션 관리나 선언적인 트랜잭션 관리에 있어 일관된 추상화를 제공.

❑ Web

- Web
  - 기본적인 웹 기반을 위한 기능을 제공한다.
  - 다중 FileUpload 처리, 리스너 와 웹 기반 application context를 위한 IoC컨테이너 초기화
- Web-Servlet
  - 웹 어플리케이션 구현을 위한 Spring MVC(Model-View-Controller) 제공
  - Struts, Webwork와 같은 프레임워크의 통합을 지원
- Web-WebSocket
  - JSR-356 스펙에 따라 구현된 WebSocket 프로그래밍 지원 모듈
- Web-Portlet
  - 포털기반의 MVC 구현을 위한 모듈 제공.

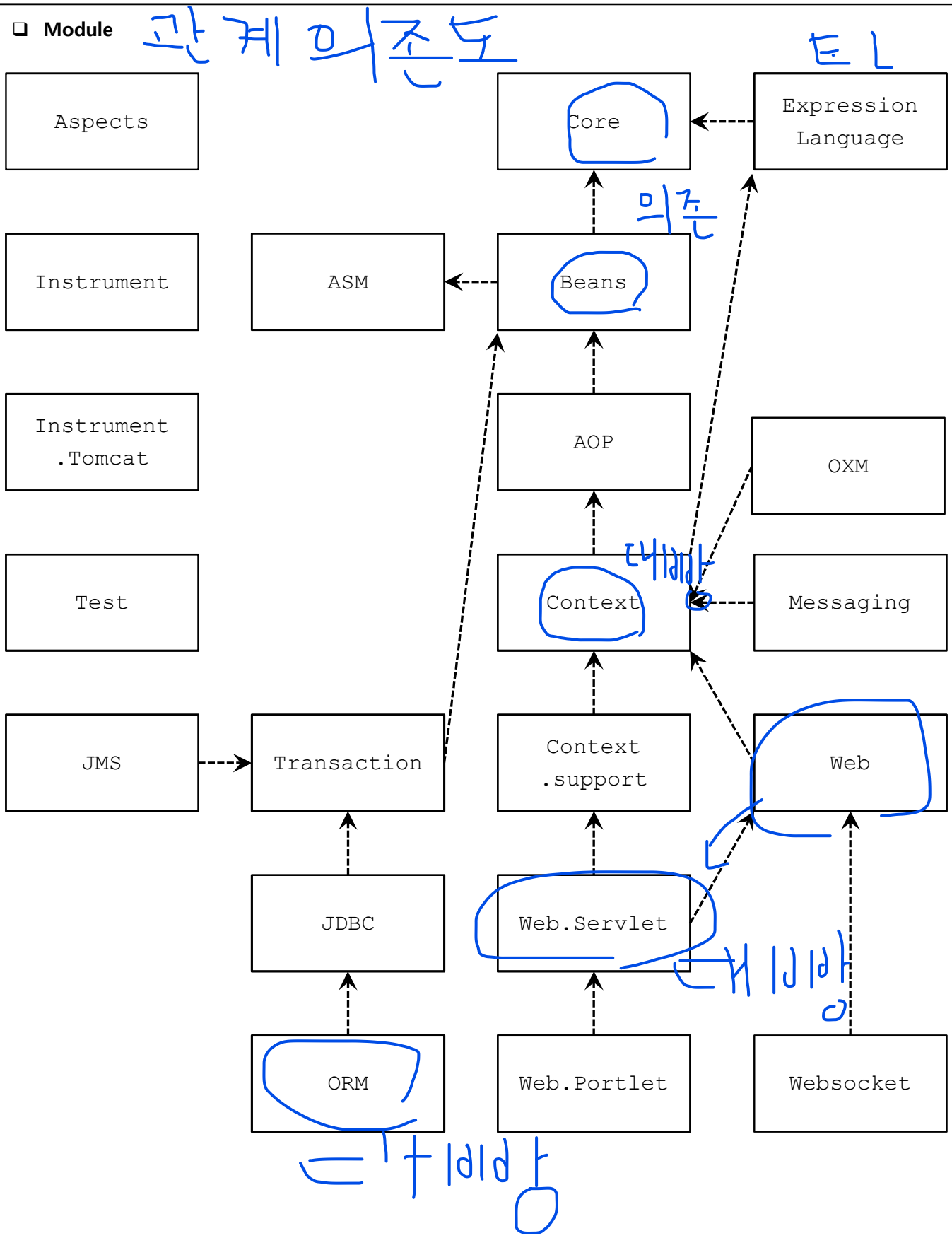
❑ Messaging

- Messaging
  - Spring-Integration 모듈에서 Message, MessageChannel, MessageHandler 등 endpoint에서 메시지 매핑을 지원하는 API 들을 스프링 기본 모듈로 이관.

❑ Test

- Test
  - JUnit이나 TestNG와 같이 Spring Componets 를 테스트하는 모듈 제공한다.
  - Mock object를 통하여 독립된 환경에서 코드를 테스트 할 수 있도록 제공한다.

❑ Module


Aspects
Core
Expression Language
Instrument
ASM
Beans
Instrument .Tomcat
AOP
OXM
Test
Context
Messaging
JMS
Transaction
Context .support
Web
JDBC
Web.Servlet
ORM
Web.Portlet
Websocket

❑ Spring 3.x 의 주요 특징

- 어노테이션 지향의 개발 모델 지원 (JSR-330, 303), 강력한 어노테이션 컴포넌트 모델(합성 어노테이션 지원)
- 다양한 기능을 가진 SpEL
- 포괄적인 REST 개발 방법 지원
- Servlet 3.0 스펙 지원 (비동기 MVC 처리, web.xml 대신 ServletContainerInitializer)
- JPA2.0 지원
- 유효성 검증(JSR-303)과 데이터 포맷팅(MessageConverter 및 Conversion Service 등)의 새로운 방식
- Scheduling 및 Caching 을 위한 새로운 전략
- Spring-test 확장
  - <jdbc: /> 네임스페이스를 통한 임베디드 데이터베이스 활용
  - @Configuration, @ActiveProfiles, @WebAppConfiguration(@ApplicationConfiguration과 함께 사용)등의 어노테이션으로 스프링과 테스트 환경 설정 간소화
  - @ContextHierarchy로 테스트시 컨테이너 설정들간의 임의의 계층구조 형성 지원.
  - Spring MVC Test 모듈 지원

❑ Spring 4.X의 새로운 특징

- Java SE 6 이상, Java EE 6 이상(Servlet 3.x 중심으로 2.5 호환가능)
- Deprecated 구성요소의 제거(패키지 및 메소드와 필드까지 포함)
- Java 8의 언어 및 API 특성 지원
  - 람다 표현식 및 메소드 레퍼런스 지원
  - JSP-310 지원 (Date 및 Time 컨버터 제공)
  - @Repeatable 어노테이션등의 지원으로, 어노테이션 반복 선언이 가능.
  - 파라미터명 리플렉션 가능(java.lang.reflect.Parameter)
- @Confitional 과 Condition 인터페이스 등의 지원으로 조건 및 상황에 따라 선택적 빈 등록 가능
  - 기존의 @Profile 어노테이션에 메터 어노테이션으로 @Conditional 을 채택하고, Spring-boot 에 다양한 Condition 구현체가 정의되는 등 광범위하게 사용됨.
  - 메타 어노테이션으로 사용하여 Composable 어노테이션을 선언하면 다양한 확장 포인트를 가질 수 있음.
- Composable 어노테이션의 속성을 통해 메타 어노테이션의 속성 재정의가 가능해짐(value 제외).
- @Lazy 정책 변경 : lazy 빈 선언 방식의 변화
  - 기존의 lazy 빈의 메타데이터로 lazy-init 이나 @Lazy를 사용했던 것과 다르게, 빈이 주입되는 지점에 @Autowired와 같은 주입 어노테이션과 함께 사용하여 lazy빈을 선언할 수 있음.
  - 일반적으로 스코프가 다른 빈을 주입하는 경우에 lazy 빈 설정을 사용하며, 이때 타겟 빈의 클래스를 대상으로 프록시가 생성됨.(CGLIB 대신 Objenesis 사용)
- Generic Type 기준의 DI 완성
- 타겟 클래스로 프록시를 생성시 CGLIB 사용으로 발생하는 문제들을 Objenesis 를 사용으로 해결
- Spring-messaging 모듈을 기본 모듈로 통합
- WebSocket 지원 확대 및 STOMP(SimpelTextOrientedProtocol) 지원
- ListenableFuture 에 기반한 AsynchRestTemplate 추가
- 다양한 ResourceResolver 구현체 제공으로 포괄적인 리소스 핸들링 지원
- Jcache(JSR-107) 기반의 캐싱 지원
- 다양한 테스팅 지원 기능 확장
- 참고
  - `http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#spring-whats-new
  - http://www.slideshare.net/sbrannen/spring-framework-40-to-41

❑ Java 8 의 새로운 특징

❑ IoC(Inversion of Control)이란?

- 실행적인 측면
  - 개발자가(만든 어플리케이션) 객체의 생성과 제거를 제어(간단한 프로그램 또는 예전 프로그램)
  - 현재 대부분의 애플리케이션은 EJB container, Web container 등의 컨테이너 기반 하에서(생성과 제거) 동작한다.
- 객체간의 관계적인 측면
  - 애플리케이션은 여러 비즈니스로직을 수행하기 위해 둘 이상의 객체를 이용한다. 즉, 각 객체는 서로 의존적으로 객체참조를 통해서 비즈니스 로직을 처리한다. 높은 결합도를 가지게 된다.
  - IoC는 객체가 필요로하는 객체를 내부에서 생성하지 않고 외부에서 주입(Inject)을 받아서 사용하는 것을 말한다. (내부의 필요한 객체 생성을 외부로부터 취득함을 의미)

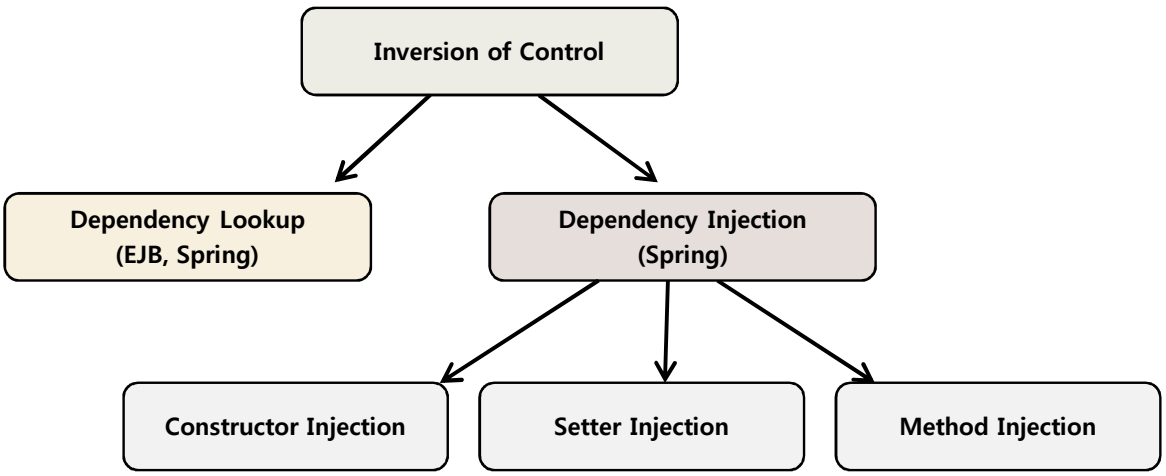
❑ DI(Dependency Injection)이란?

- Dependency Injection이란 모듈간의 의존성을 모듈의 외부(컨테이너)에서 주입시켜주는 기능으로 Inversion of Control의 한 종류이다.
- 런타임시 사용하게 될 의존대상과의 관계를 Spring Framework 이 총체적으로 결정하고 그 결정된 의존특징을 런타임시 부여한다.

2004년초, Martin Fowler는 그의 사이트 독자들에게 물었다.
"the question is, what aspect of control are [they] inverting?'
"질문은, 제어의 측면에서 무엇이 역행(전)하는가?"
Martin은 Inversion of Control 용어에 대해서 설명한 후 패턴 이름을 바꾸거나 적어도 더 나은 의미 있는 이름을 제안했고, Dependency Injection 용어를 사용하기 시작한다.
http://martinfowler.com/articles/injection.html
http://javacan.tistory.com/entry/120(번역본)

❑ IoC의 분류

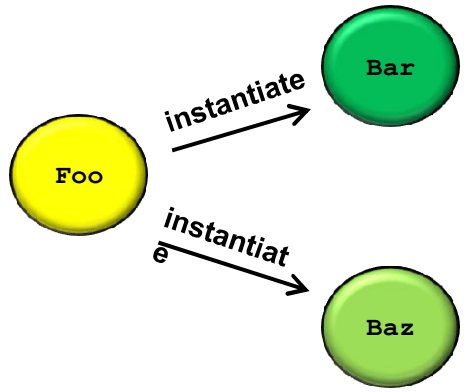
- 일부 다르게 표현되지만 일반적으로 다음처럼 분류한다.



- DL : 저장소에 저장되어 있는 Bean에 접근하기 위하여 컨테이너에서 제공하는 API를 이용하여 사용하고자 하는 Bean을 Lookup하는 것을 말한다.
- DI : 각 클래스 사이의 의존관계를 빈 설정 정보를 바탕으로 컨테이너가 자동적으로 연결해주는 것을 말한다. 개발자들은 빈 설정 파일(저장소 관리 파일)에 의존관계가 필요하다는 정보를 추가하면 된다.
  1. Constructor Injection : 생성자를 이용하여 의존관계를 연결
  2. Setter Injection : setter 메소드를 이용하는 의존관계를 연결
  3. Method Injection : Setter / Constructor Injection이 가지고 있는 한계점을 극복하기 위하여 Spring에서 지원하고 있는 DI의 한 종류. Singleton/ Non Singleton 인스턴스의 의존관계를 연결 할 때 사용

❑ Non-IoC/DI vs IoC/DI

Non-IoC/DI

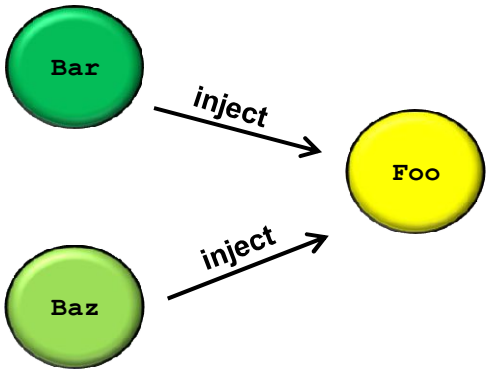



```
package x.y;

public class Foo {

  public void process() {
      Bar bar = new Bar();
      Baz baz = new Baz();
      // ...
  }
}
```

IoC/DI




```
package x.y;
public class Foo {
   public Foo(Bar bar, Baz baz) {
     // ...
  }
}
<beans>
  <bean name="foo" class="x.y.Foo">
      <constructor-arg>
         <bean class="x.y.Bar"/>
      </constructor-arg>
      <constructor-arg>
         <bean class="x.y.Baz"/>
      </constructor-arg>
   </bean>
</beans>
```

❑ Container

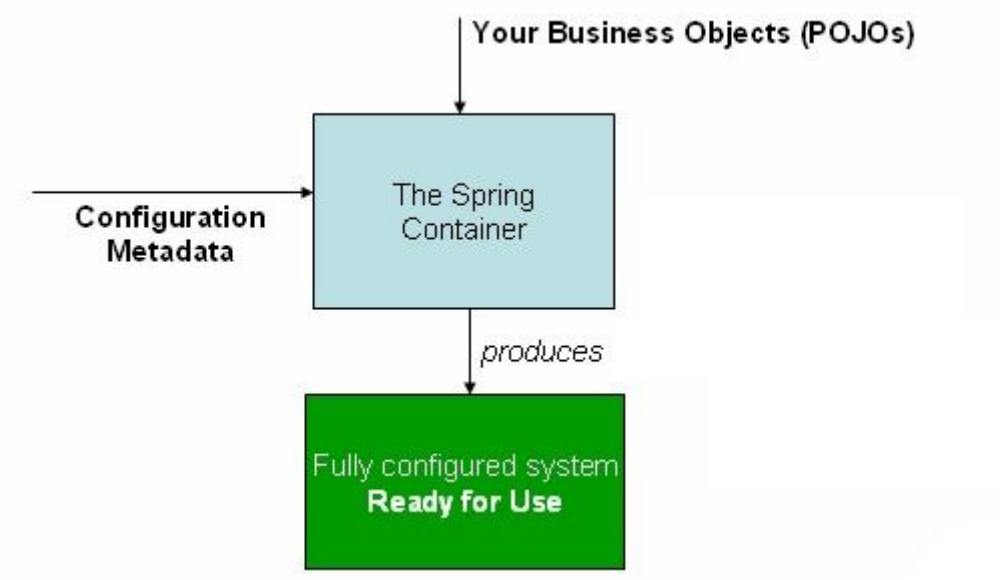
- Spring IoC Container는 객체를 생성하고, 객체 간의 의존성을 이어주는 역할을 한다.
- bean을 포함하고 관리하는 책임을 지는 Spring IoC의 실제
- 설정 메타데이터로 bean을 관리한다.
  (ex, XML , annotation, Java)

❑ 설정 정보(Configuration Metadata)

- Spring IoC container가 "객체를 생성하고, 객체간의 의존성을 이어줄 수 있도록" 필요한 정보를 제공한다.
  설정 정보는 XML 형태, Java Annotation 형태, Java 파일 자체 등을 이용하여 설정한다.

❑ Bean

- Spring IoC컨테이너에 의해 관리되는 객체
- bean은 전형적으로 인스턴스화되는 객체
- bean들과 각각에 대한 의존성은 설정 메타데이터로 반영



❑ Spring IoC Container 유형

- BeanFactory 인터페이스
  - BeanFactory 인터페이스는 Spring IoC Container의 기능을 정의하고 있는 기본 인터페이스이다.
  - Bean 생성 및 의존성 주입, 생명주기 관리 등의 기능을 제공한다.
- ApplicationContext 인터페이스
  - BeanFactory를 구현하는 상위개념
  - Spring AOP, 메시지 지원, 국제화 지원, 이벤트 지원, 리스너 지원 등 애플리케이션 구현에 적합한 컨테이너
  - 모든 ApplicationContext 구현체는 BeanFactory의 기능을 모두 제공하므로, 특별한 경우를 제외하고는 ApplicationContext를 사용하는 것이 바람직하다.
- WebApplicationContext 인터페이스
  - ApplicationContext기반으로 web 애플리케이션을 위한 추가적인 기능을 제공하는 컨테이너

- BeanFactory는 getBean 메서드가 호출 할 때까지 Bean의 생성을 미룬다(Lazy loading) 그에 반해 ApplicationContext는 Context를 시작할 때 모든 Singleton Bean을 미리 로딩한다. (default-lazy-init으로 제어)

❑ Container 초기화

- BeanFactory 인터페이스를 구현한 대표적인 클래스
  - org.springframework.bean.factory.xml.XmlBeanFactory

```
Resource resource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

```
ClassPathResource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);}
```

- ApplicationContext 인터페이스를 구현한 클래스
  - org.springframework.context.support.ClassPathXmlApplicationContext
  - org.springframework.context.support.FileSystemXmlApplicationContext
  - org.springframework.context.support.GenericXmlApplicationContext(상위 둘의 혼합형으로 유연한 컨텍스트 생성이 가능하고, refresh메소드를 통해 컨텍스트 리로딩이 가능함.)
  - org.springframework.context.annotation.AnnotationConfigApplicationContext (Spring3.x부터 자바 기반의 Configuration 정의를 지원하고 있는데, 순수 자바 코드로만 빈 메타데이터를 비롯한 기본 설정등을 정의하는 경우의 컨테이너 타입)

```
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"services.xml", "daos.xml"});
ApplicationContext context = new GenericXmlApplicationContext("classpath:daos.xml");
ApplicationContext context = new GenericXmlApplicationContext("file:d:/config/daos.xml");
ApplicationContext context = new AnnotationConfigApplicationContext(SampleConfig.class);
```

- WebApplicationContext 인터페이스를 구현한 클래스
  - org.springframework.web.context.support.XmlWebApplicationContext

❑ 설정 metadata

- 설정 메타데이터를 사용하여 Bean을 정의한다. XML 기반의 메타데이터를 가장 많이 사용한다.
- XML 형태 외에 Propetries 파일, Java파일 형태로도 설정 메타데이터를 설정할 수 있다.

- XML 설정 파일은 <beans/> element를 root로 갖는다. 아래는 기본적인 XML 설정 파일의 모습이다.

```
<?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-3.0.xsd">
   <bean id="..." class="...">
      <!-- collaborators and configuration for this bean go here -->
   </bean>

   <!-- more bean definitions go here -->

</beans>
```

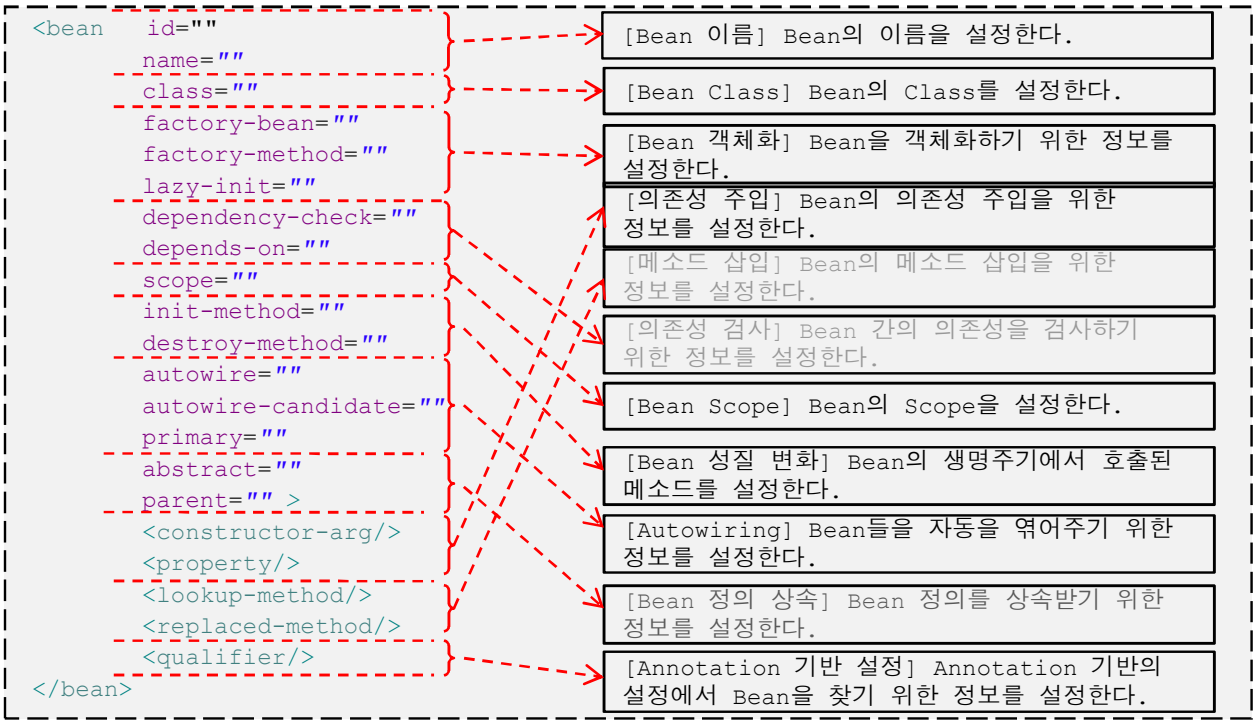
- XML 설정은 여러 개의 파일로 구성될 수 있으며, <import/> element를 사용하여 다른 XML 설정 파일을 import 할 수 있다.
- <import />를 <bean /> 이전에 설정한다.
  - Spring 3.x 부터 ${propertyName}형태의 프로퍼티 접근 시 JVM 시스템 속성이나 시스템 환경변수 이외의 커스텀 속성들까지 PropertySource 를 통해 관리되는 다양한 프로퍼티들에 접근할 수 있게 됨.

```
<beans>
    <import resource="{customer}-cofig.xml"/>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

❑ Bean 정의

- Bean 정의는 Bean을 객체화하고 의존성을 주입하는 등의 관리를 위한 정보를 담고 있다. XML 설정에서는 <bean/> element가 Bean 정의를 나타낸다. Bean 정의는 아래와 같은 속성을 가진다.



❑ Bean 이름

- 모든 Bean은 하나의 id를 가지며, 하나 이상의 name을 가질 수 있다. id는 container 안에서 고유해야 한다.
- bean에 대한 명명규칙은 camel-cased 표기법을 따르는 것이 좋다.
- name 속성, 공백으로 별칭 사용가능
- id가 명확하지만 가끔 "/" 같은 특별한 문자를 id 속성에서는 사용이 안됨, 그런 경우 name속성 사용

```
<bean id="exampleBean" class="example.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

- Bean은 <alias/> element를 이용하여 추가적인 name을 가질 수 있다.

```
<alias name="fromName" alias="toName"/>
```

❑ Bean 객체화

- 생성자를 이용한 객체화
  - 일반적으로 Bean 객체화는 Java 언어의 'new' 연산자를 사용한다. 이 경우 별도의 설정은 필요없다.

```
<bean id="exampleBean" class="example.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

- static factory 메서드를 사용한 객체화
  - 'new' 연산자가 아닌 static factory 메소드를 사용하여 Bean을 객체화할 수 있다.

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance" />
```

- 다른 Bean의 factory 메서드를 사용한 객체화
  - 별도의 Factory 클래스의 메소드를 사용하여 Bean을 객체화할 수 있다.
  - class 속성은 사용하지 않고 factory-bean 속성에 해당 bean의 이름을 명시

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">
   <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="serviceLocator"
      factory-method="createInstance" />
```

❑ Lazy Instantiation

- <bean/> element의 'lazy-init' attribute를 사용하여 Bean 객체화 시기를 설정할 수 있다.
- 일반적으로 Bean 객체화는 BeanFactory가 객체화되는 시점에 수행된다. 만약, 'lazy-init' attribute 값이 'true'인 경우, 설정된 Bean의 객체가 실제로 필요하다고 요청한 시점에 객체화가 수행된다.
- 'lazy-init' attribute가 설정되어 있지 않으면 기본값을 사용한다. Spring Framework의 기본값은 'false'이다.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

- <beans/> element의 'default-lazy-init' attribute를 사용하여 XML 설정 파일 내의 모든 Bean 정의에 대한 lazy-init attribute의 기본값을 설정할 수 있다.

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

❑ Dependency Injection

- 의존성 주입에는 Constructor Injection과 Setter Injection의 두가지 방식이 있다.

❑ Constructor Injection

- Constructor Injection은 argument를 갖는 생성자를 사용하여 의존성을 주입하는 방식이다. <constructor-arg/> element를 사용한다.
- 생성자의 argument와 <constructor-arg/> element는 class가 같은 것끼리 매핑한다.

```
package x.y;
public class Foo {
public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

```
<beans>
   <bean name="foo" class="x.y.Foo">
      <constructor-arg name="bar">
         <bean class="x.y.Bar"/>
      </constructor-arg>
      <constructor-arg name="baz">
         <bean class="x.y.Baz"/>
      </constructor-arg>
   </bean>
</beans>
```

- 만약 생성자가 같은 class의 argument를 가졌거나 primitive type인 경우 argument와 <constructor-arg/> element간의 매핑이 불가능하다. 이 경우, Type을 지정하거나 순서를 지정할 수 있다.

```
package examples;

public class ExampleBean {
   // No. of years to the calculate the Ultimate Answer
   private int years;
   // The Answer to Life, the Universe, and Everything
   private String ultimateAnswer;

   public ExampleBean(int years, String ultimateAnswer) {
      this.years = years;
      this.ultimateAnswer = ultimateAnswer;
   }
}
```

- Type 지정

```
<bean id="exampleBean" class="examples.ExampleBean">
   <constructor-arg type="int" value="7500000"/>
   <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

- 순서 지정

```
<bean id="exampleBean" class="examples.ExampleBean">
   <constructor-arg index="0" value="7500000"/>
   <constructor-arg index="1" value="42"/>
</bean>
```

❑ Setter Injection

- Setter Injection은 argument가 없는 기본 생성자를 사용하여 객체를 생성한 후, setter 메소드를 사용하여 의존성을 주입하는 방식으로, <property/> element를 사용한다.
- Class에 attribute(또는 setter 메소드 명)과 <property/> element의 'name' attribute를 사용하여 매핑한다.

```
public class ExampleBean {
    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }
    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested <ref/> element -->
    <property name="beanOne"><ref bean="anotherExampleBean"/></property>
    <!-- setter injection using the neater 'ref' attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

- 매핑 규칙은 <property/> element의 'name' attribute의 첫문자를 알파벳 대문자로 변경하고 그 앞에 'set'을 붙인 setter 메소드를 호출한다.

```
<bean id="exampleBean"
      class="examples.ExampleBean">
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>
</bean>
```

```
public class ExampleBean {
    public void setBeanOne(
                AnotherBean beanOne)
    {
        this.beanOne = beanOne;
    }
}
```

❑ 의존성 설정 상세

- <constructor-arg/> element 과 <property/> element는 '명확한 값', '다른 Bean에 대한 참조', 'Inner Bean', 'Collection', 'Null' 등의 값을 가질 수 있다.
- <value /> : 명확한 값
  - Java Primitive Type, String 등의 명확한 값을 나타낸다. 사람이 인식 가능한 문자열 형태를 값으로 갖는 <value/> element를 사용한다. Spring IoC container가 String 값을 해당하는 type으로 변환하여 주입해준다.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
   <!-- results in a setDriverClassName(String) call -->
   <property name="driverClassName">
      <value>com.mysql.jdbc.Driver</value>
   </property>
</bean>
```

- <ref /> : 다른 빈을 참조
  - <ref/> element를 사용하여 다른 Bean 객체를 참조할 수 있다. 참조할 객체를 지정하는 방식은 bean', 'local', 'parent' 등 이 있다.
  - bean : 가장 일반적인 방식으로 같은 컨테이너 또는 부모컨테이너에서 bean에 대한 객체를 찾는다.
    속성값은 대상 bean의 'id' 또는 'name' 속성 중 하나

```
<ref bean="someBean"/>
```

  - local : 같은 XML 설정 파일 내에서 bean 객체를 찾는다.
    속성값은 대상 bean의 'id'이어야만 함

```
<ref local="someBean"/>
```

  - parent : 부모 XML 설정 파일 내에 bean 애 대한 객체를 찾는다
    속성값은 대상 bean의 'id' 또는 'name' 속성 중 하나

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
   <!-- insert dependencies as required as here -->
</bean>
```

O

```
<!-- in the child (descendant) context -->
<bean id="accountService"
class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
            <ref parent="accountService"/>
      </property>
</bean>
```

X

❑ 의존성 설정 상세

- <bean /> : Inner Bean
  - <property/> 또는 <constructor-arg/> element 안에 있는 <bean/> element를 inner bean이라고 한다. Inner bean의 'scope' flag 와 'id', 'name'은 무시된다. Inner bean의 scope은 항상 prototype이다. 따라서 inner bean을 다른 bean에 주입하는 것은 불가 능하다.

```
<bean id="outer" class="...">
   <!-- instead of using a reference to a target bean, simply define the target
bean   inline -->
   <property name="target">
      <bean class="com.example.Person"> <!-- this is the inner bean -->
         <property name="name" value="Fiona Apple"/>
         <property name="age" value="25"/>
      </bean>
   </property>
</bean>
```

- Collection
  - Java Collection 타입인 List, Set, Map, Properties를 표현하기 위해 <list/>, <set/>, <map/>, <props/> element가 사용된다.
  - map의 key와 value, set의 value의 값은 아래 element 중 하나가 될 수 있다.

```
bean | ref | idref | list | set | map | props | value | null
```

```
<!-- results in a setAdminEmails(java.util.Properties) call -->
   <property name="adminEmails">
      <props>
         <prop key="administrator">administrator@example.org</prop>
         <prop key="support">support@example.org</prop>
         <prop key="development">development@example.org</prop>
      </props>
   </property>
   <!-- results in a setSomeList(java.util.List) call -->
   <property name="someList">
      <list>
         <value>a list element followed by a reference</value>
         <ref bean="myDataSource" />
      </list>
   </property>
```

- Collection

```
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
   <map>
      <entry>
         <key><value>an entry</value></key>
         <value>just some string</value>
      </entry>
      <entry>
         <key><value>a ref</value></key>
         <ref bean="myDataSource" />
      </entry>
   </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
   <set>
      <value>just some string</value>
      <ref bean="myDataSource" />
   </set>
</property>
```

- <null />
  - Java의 null 값을 사용하기 위해서 <null/> element를 사용한다.
  - Spring IoC container는 value 값이 설정되어 있지 않은 경우 빈 문자열("")로 인식한다.

```
<bean class="ExampleBean">
   <property name="email"><value/></property>
</bean>
```

  - 위 ExampleBean의 email 값은 ""이다. 아래는 email의 값이 null인 예제이다.

```
<bean class="ExampleBean">
   <property name="email"><null/></property>
</bean>
```

- Compound property name
  - 복합 형식의 property 이름도 사용할 수 있다.
  - 마지막 property 속성을 제외한 나머지는 null이 아니어야 한다.
  - 아래 예시에서 id="foo" 빈이 객체화 될 때 foo의 속성 fred , fred의 속성 bob 은 null이 아니어야 한다.

```
<bean id="foo" class="foo.Bar">
   <property name="fred.bob.sammy" value="123" />
</bean>
```

❑ 설정 메타데이터 간략화

- <property/>, <constructor-arg/>, <entry/> element의 <value/> element는 'value' attribute로 대체될 수 있다.

```
<property name="myProperty">
   <value>hello</value>
</property>
```

=

```
<property name="myProperty"
value="hello"/>
```

```
<constructor-arg>
   <value>hello</value>
</constructor-arg>
```

=

```
<constructor-arg value="hello"/>
```

- <property/>, <constructor-arg/> element의 <ref/> element는 'ref' attribute로 대체될 수 있다.

```
<property name="myProperty">
   <ref bean="myBean">
</property>
```

=

```
<property name="myProperty"
          ref="myBean"/>
```

```
<constructor-arg>
   <ref bean="myBean">
</constructor-arg>
```

=

```
<constructor-arg ref="myBean"/>
```

- <entry/> element의 'key', 'ref' element 는 'key-ref', 'value-ref' attribute로 대체될 수 있다.

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

=

```
<entry key-ref="myKeyBean"
       value-ref="myValueBean"/>
```

❑ 설정 메타데이터 간략화(계속)

- p-namespace
  - <property/> element 대신 'p-namespace'를 사용하여 XML 설정을 작성할 수 있다. 아래 classic bean과 p-namespace bean은 동일한 Bean 설정이다.

```
<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"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean name="classic" class="com.example.ExampleBean">
      <property name="email" value="foo@bar.com/>
   </bean>

   <bean name="p-namespace" class="com.example.ExampleBean"
         p:email="foo@bar.com"/>
</beans>
```

  - Attribute 이름 끝에 '-ref'를 붙이면 Bean 참조로 인식한다.

```
   <bean name="john-classic" class="com.example.Person">
      <property name="name" value="John Doe"/>
      <property name="spouse" ref="jane"/>
   </bean>
   <bean name="john-modern"  class="com.example.Person"
         p:name="John Doe"
         p:spouse-ref="jane"/>
    <bean name="jane" class="com.example.Person">
     <property name="name" value="Jane Doe"/>
   </bean>
```

- c-namespace : constructor injection 지원. <constructor-arg>대신 사용.

```
package examples;

public class ExampleBean {
    private int years;
    private String ultimateAnswer;
   public ExampleBean(int years, String ultimateAnswer) {
      this.years = years;
      this.ultimateAnswer = ultimateAnswer;
   }
}
```

```
<bean id="exampleBean" class="examples.ExampleBean">
   <constructor-arg type="int" value="7500000"/>
   <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

```
<bean id="exampleBean"  class="examples.ExampleBean"
      c:years="7500000"
      c:ultimateAnswer="42"
/>
```

❑ 설정 정보의 외부화

- PropertyPlaceholderConfigurer 를 사용하여 외부 Property 파일로부터 속성을 로딩하고 변수를 채운다.

```
# com/foo/jdbc.properties
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
```

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>
<bean id="dataSource" destroy-method="close"
       class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
</bean>

<!--  Spring 2.5 부터 제공하는 context 네임스페이스를 이용한 방법
      locations 속성을 이용하는 경우 콤마(",")를 이용해 여러 파일 로딩 가능
       PropertySourcePlaceholderConfigurer의 등록  -->
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

<!--  Spring 2.0 부터 제공하는 util 네임스페이스를 이용한 방법
      properties 객체를 빈으로 등록해 재활용이 가능하도록 함.-->
<util:properties id="jdbcProps" location="classpath:com/foo/jdbc.properties"/>

<bean id="dataSource" destroy-method="close"
       class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="#{jdbcProps['jdbc.driverClassName']}"/>
  <property name="url" value="#{jdbcProps['jdbc.url']}"/>
  <property name="username" value="#{jdbcProps['jdbc.username']}"/>
</bean>
```

- PropertySource는 키-값의 쌍으로 구성된 속성들의 추상화로 Spring 3.x 부터 추가된 Environment 추상화를 통해 우선순위가 결정된 PropertySource 를 순차적으로 접근할 수 있게됐다. Spring 의 DefaultEnvironment 는 다음과 같은 PropertySource를 기본적으로 구성하고 있다.
  - JVM 시스템 속성값의 집합(System.getProperties()와 같은 방식)
  - 시스템 환경변수의 집합(System.getEnv()와 같은 방식)
  - 우선순위를 설정하게 커스텀 프로퍼티 소스를 추가할 수 있음.

```
ConfigurableApplicationContext ctx = new GenericXmlApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

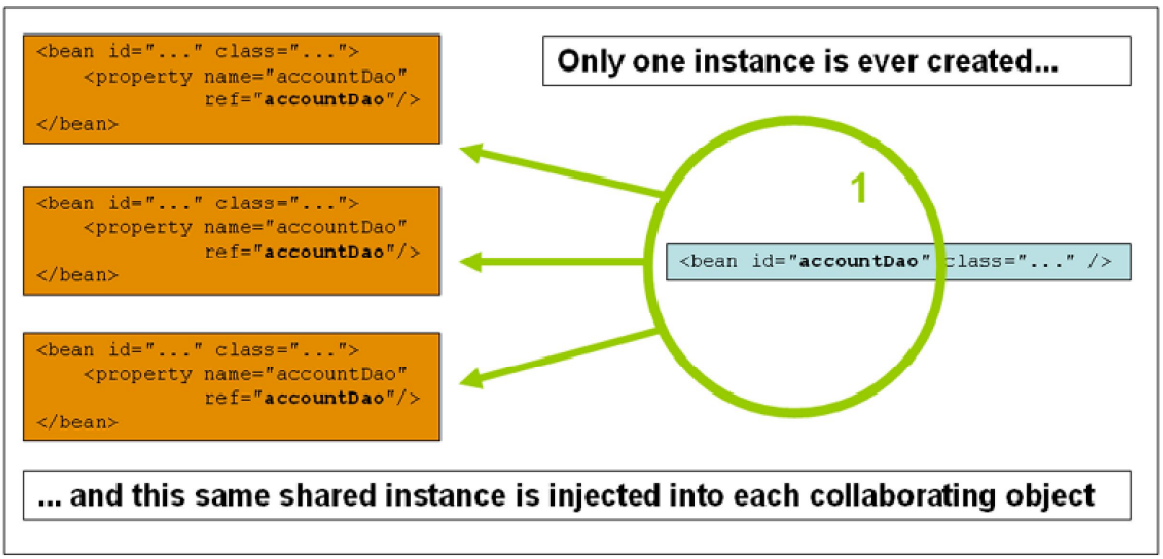
이렇게 추가된 PropertySource 는 PropertySourcePlaceholderConfigurer를 beanFactoryPostProcessor 로 등록하면 프로퍼티 주입시에 ${propertyName }과 같은 형식으로 주입이 가능하다.
일반적으로 PropertySourcePlaceholderConfigurer의 등록할때는 <context:place-placeholder > 를 사용한다.

❑ Bean Scopes

- Bean Scope은 객체가 유효한 범위로 기본적으로 5가지의 scope이 있다
- 이외에 사용자 정의 scope 도 있다.(레퍼런스 참조)

Scope	설 명
singleton	컨테이너는 오직 한 개의 빈만을 생성하여 관리한다. Spring은 기본적으로 모든빈을 singleton 으로 관리.
prototype	매번 요청에 대하여 bean의 인스턴스를 생성 생성 상태유지를 해야 하는 빈은 prototyp으로 선언해야 한다.
request	하나의 HTTP request의 생명주기 안에 단 하나의 객체만 존재한다. Web-aware Spring ApplicationContext 안에서만 유효하다.
session	하나의 HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. Web-aware Spring ApplicationContext 안에서만 유효하다.
global session	하나의 global HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. 일반적으로 portlet context 안에서 유효하다. Web-aware Spring ApplicationContext 안에서만 유효하다.

- Singleton Scope
  - Bean이 singleton인 경우, 단 하나의 객체만 공유된다.

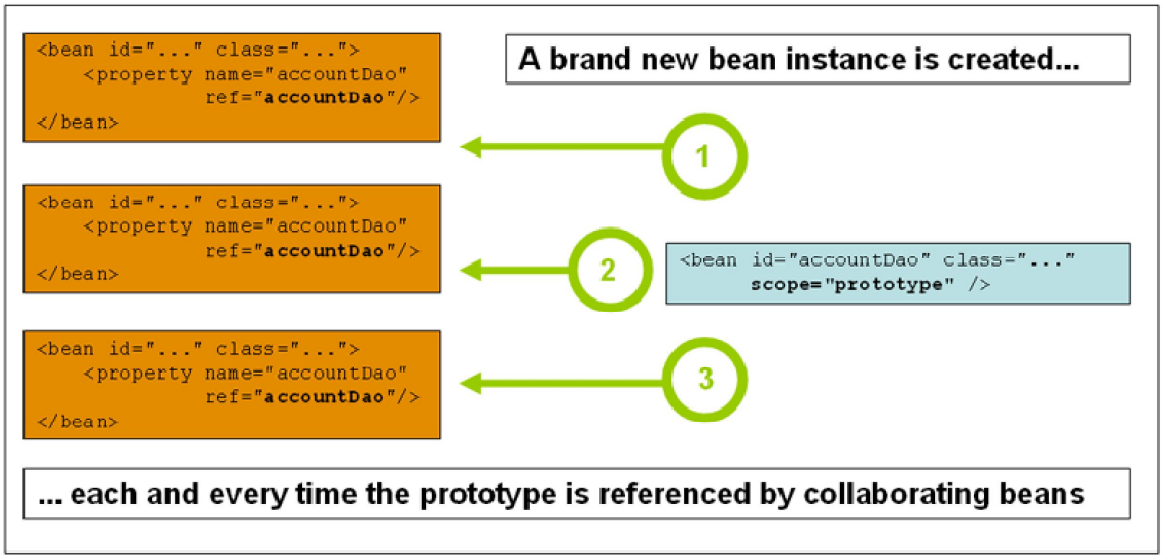


  - Spring의 기본 scope은 'singleton'이다. 설정하는 방법은 아래와 같다.

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

❑ Bean Scopes

- Prototype Scope
  - Singleton이 아닌 prototype scope의 형태로 정의된 bean은 필요한 매 순간 새로운 bean 객체가 생성된다



  - 설정하는 방법은 아래와 같다.

```
<bean id="accountService" class="com.foo.DefaultAccountService"
      scope="prototype"/>
```

- Method Injection
  - Dependency Injection을 사용하는 경우 Singleton Bean 과 Non-Singleton Bean(prototype) 과 의존성을 가질 수 있다. 이 같은 상황이 발생했을 때
    1. BeanFactoryAware를 구현하는 방법은 Spring API에 종속적이므로 권장하지 않음
    2. Lookup Method Injection <lookup-method />를 이용해 참조 관계 정의
    3. Method Replacement <replaced-method />를 이용해 기존의 메서드를 변경하지 않는 상태에서 메서드의 기능을 변경하고자 할 때
  - Lookup Method Injection 예시

```
package fiona.apple;
public abstract class CommandManager {
  public Object process(Object commandState) {
   // grab a new instance of the appropriate Command interface
   Command command = createCommand();
   command.setState(commandState);
   return command.execute();
  }
  // okay... but where is the implementation of this method?
  protected abstract Command createCommand();
}
```

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
  <!-- inject dependencies here as required -->
</bean>
<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
  <lookup-method name="createCommand" bean="command"/>
</bean>
```

❑ Bean Scopes

- 기타 Scope
  - request, session, global session은 Web 기반 ApplicationContext만 유효한 scope이다.
  - Request Scope

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

위 정의에 따라, Spring container는 모든 HTTP request에 대해서 'loginAction' bean 정의에 대한 LoginAction 객체를 생성할 것이다. 즉, 'loginAction' bean은 HTTP request 수준에 한정된다(scoped). 요청에 대한 처리가 완료되었을 때, 한정된(scoped) bean도 폐기된다.

  - Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

위 정의에 따라, Spring container는 하나의 HTTP Session 일생동안 'userPreferences' bean 정의에 대한 UserPreferences 객 체를 생성할 것이다. 즉, 'userPreferences' bean은 HTTP Session 수준에 한정된다 (scoped). HTTP Session이 폐기될 때, 한 정된(scoped) bean도 폐기된다.

  - Global Session Scope

```
<bean id="userPreferences" class="com.foo.UserPreferences"
        scope="globalSession"/>
```

global session scope은 HTTP Session scope과 비슷하지만 단지 portlet-based web 어플리케이션에서만 사용할 수 있다. Portlet 명세(specifications)는 global Session을 하나의 portlet web 어플리케이션을 구성하는 여러 portlet들 모두가 공유하는 것으로 정의하고 있다. global session scope으로 설정된 bean은 global portlet Session의 일생에 한정된다.

- Scope이 다른 Bean에서 참조하는 경우 <aop:scoped-proxy />와 함께 작성한다.

```
<!-- an HTTP Session-scoped bean exposed as a proxy -->
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
      <!-- this next element effects the proxying of the surrounding bean -->
      <aop:scoped-proxy/>
</bean>

<!-- a singleton-scoped bean injected with a proxy to the above bean -->
<bean id="userService" class="com.foo.SimpleUserService">
    <!-- a reference to the proxied userPreferences bean -->
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

  - 위의 예시에서 userPerferences 빈은 scope이 session이지만 userService의 빈의 scope은 singleton이기 때문에 문제가 발생한다. 정상적으로 동작하려면 매 세션마다 userPerferences 객체가 생성되어야 하지만 userService는 객체가 한번만 생성되기 때문에 원하는 대로 동작을 하지 못한다.
  - 따라서 매 세션마다 새로운 객체를 만들어줄 Proxy를 만들기 위해 <aop:scoped-proxy />를 지정한다.

❑ Lifecycle Callback

- Spring IoC Container는 Bean의 각 생명주기에 호출되도록 설정된 메소드를 호출해준다.
  - PostConstruct PreDestory 상황을 처리하는 방법
- Initialization callback
  - org.springframework.beans.factory.InitializingBean interface를 구현하면 bean에 필요한 모든 property를 설정한 후, 초기화 작업을 수행한다. InitializingBean interface는 다음 메소드를 명시하고 있다.

```
void afterPropertiesSet() throws Exception;
```

일반적으로, InitializingBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'init-method' attribute를 사용한다.

```
public class ExampleBean {

   public void init() {
   // do some initialization work
   }
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

- Destruction callback
  - org.springframework.beans.factory.DisposableBean interface를 구현하면, container가 파괴될 때 bean이 callback를 받을 수 있다. DisposableBean interfce는 다음 메소드를 명시하고 있다.

```
void destroy() throws Exception;
```

일반적으로, DisposableBean interface의 사용을 권장하지 않는다. 왜냐하면 code가 불필요하게 Spring과 결합되기(couple) 때문이다. 대안으로, bean 정의는 초기화 메소드를 지정할 수 있다. XML 기반 설정의 경우, 'destroy-method' attribute를 사용 한다

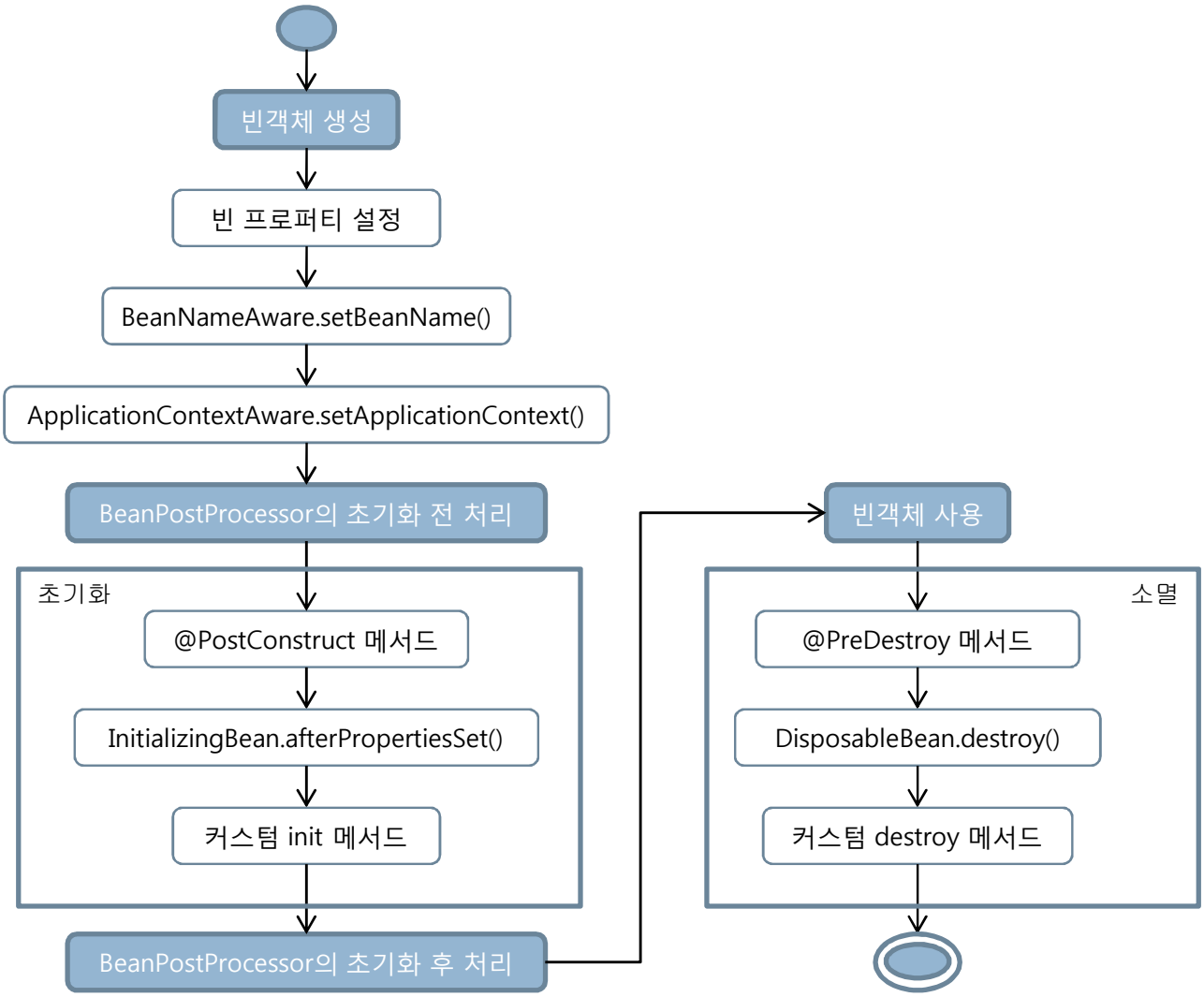
```
public class ExampleBean {
   public void cleanup() {
   // do some destruction work (like releasing pooled connections)
   }
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

- 기본 Instantiation & Destruction callback 선언하기
  - Spring IoC container 레벨에서 기본 Instantiation & Destruction callback 메소드를 지정할 수 있다. <beans/> element의 'default-init-method', 'default-destroy-method' attribute를 사용한다.

```
<beans default-init-method="init" default-destroy-method="destroy">

   <bean id="blogService" class="com.foo.DefaultBlogService">
      <property name="blogDao" ref="blogDao" />
   </bean>
</beans>
```


빈객체 생성
빈 프로퍼티 설정
BeanNameAware.setBeanName()
ApplicationContextAware.setApplicationContext()
BeanPostProcessor의 초기화 전 처리
초기화
@PostConstruct 메서드
InitializingBean.afterPropertiesSet()
커스텀 init 메서드
BeanPostProcessor의 초기화 후 처리
빈객체 사용
소멸
@PreDestroy 메서드
DisposableBean.destroy()
커스텀 destroy 메서드

❑ Annotation 기반 설정

- Spring은 Java Annotation을 사용하여 복잡한 설정으로 부터 별도의 XML정의 없이 Bean 정의를 설정할 수 있다.
- Spring 2.0 에서는 Transaction 어노테이션, Spring 2.5 에서는 의존성 관련/ MVC 어노테이션의 활성화
- Spring3.0 에서는 2.5 버전보다 추가 및 확장하여 발전시켰다.

- 표준 어노테이션(JSR-330, Dependency Injection for Java)을 지원한다.
- 이 기능을 사용하기 위해서는 다음 namespace와 element를 추가해야 한다.

```
<?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
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>

</beans>
```

- @Required
  - @Required annotation은 setter 메소드에 적용된다. @Required annotation이 설정된 property는 <property/>, <constructor-arg/> element를 통해서 명시적으로 값이 설정되거나, autowiring에 의해서 값이 설정되어야 한다.

```
public class SimpleMovieLister {

   private MovieFinder movieFinder;

   @Required
   public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
   }

   // ...
}
```

❑ Annotation 기반 설정

- @Autowired
  - @Autowired annotation은 자동으로 엮을 property를 지정하기 위해 사용한다. setter 메소드, 일반적인 메소드, 생성자, field 등에 적용된다.
  - Setter 메소드

```
public class SimpleMovieLister {

   private MovieFinder movieFinder;

   @Autowired
   public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
   }
   // ...
}
```

  - 일반적인 메소드

```
public class MovieRecommender {

   private MovieCatalog movieCatalog;

   private CustomerPreferenceDao customerPreferenceDao;

   @Autowired
   public void prepare(MovieCatalog movieCatalog,
                       CustomerPreferenceDao customerPreferenceDao) {
      this.movieCatalog = movieCatalog;
      this.customerPreferenceDao = customerPreferenceDao;
   }
   // ...
}
```

  - 생성자 및 field

```
public class MovieRecommender {

   @Autowired
   private MovieCatalog movieCatalog;

   private CustomerPreferenceDao customerPreferenceDao;

   @Autowired
   public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
      this.customerPreferenceDao = customerPreferenceDao;
   }
   // ...
}
```

❑ Annotation 기반 설정

- @Qualifier
  - @Autowired annotation만을 사용하는 경우, 같은 Type의 Bean이 둘 이상 존재할 때 문제가 발생한다. 이를 방지하기 위해서 @Qualifier annotation을 사용하여 찾을 Bean의 대상 집합을 좁힐 수 있다. @Qualifier annotation은 field 뿐 아니라 생성자 또 는 메소드의 parameter에도 사용할 수 있다.
  - Field

```
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
    // ...
}
```

  - 메소드 Parameter

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
                        CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

  - @Qualifier annotation의 값으로 사용되는 qualifier는 <bean/> element의 <qualifier/> element로 설정한다.

```
    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- inject any dependencies required by this bean -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>
```

❑ Annotation 기반 설정

- @Inject
  - Spring, Google guice 의 제안으로 채택된 JSR-330 어노테이션.
  - 타입 기반의 자동 주입을 지원해주는 어노테이션으로, 필드, setter, 생성자, 메소드에 사용가능.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

- 이름 기반의 주입방식과 동시 사용하고 싶은 경우

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    @Named("myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

- @Inject 는 JSR-330 스펙의 Provider 를 통해 지연 주입 혹은 멀티 인스턴스 주입이 가능하다.
  - 주입되는 빈의 scope 가 prototype 이고 싱글턴 빈에서 이를 주입받아야 하는 경우 유용함.

```
public class SimpleMovieLister {

    private Set<MovieFinder> finderSet;

    @Inject
    @Named("myMovieFinder")
    public void setMovieFinders(Provider<MovieFinder> finderProvider) {
        this.finderSet = new HashSet<MovieFinder>();
        this.finderSet.add(finderProvider.get());
    }
    // ...
}
```

Annotation	@Autowired	@Inject	@Resource
Injection 방식	type-driven injection	type-driven injection	name-matching injection
사용가능한 위치	멤버변수, setter 메소드, 생성자, 일반 메소드	멤버변수, setter 메소드, 생성자, 일반 메소드	멤버변수, setter 메소드

❑ Annotation 기반 설정

- @Resource
  - @Resource annotation의 name 값으로 대상 bean을 찾을 수 있다. @Resource annotation은 field 또는 메소드에 사용할 수 있다.

```
public class SimpleMovieLister {

   private MovieFinder movieFinder;

   @Resource(name="myMovieFinder")
   public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
   }
}
```

  - @Resource annotation에 name 값이 없을 경우, field 명 또는 메소드 명을 이용하여 대상 bean을 찾는다.

```
public class SimpleMovieLister {

   private MovieFinder movieFinder;

   @Resource
   public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
   }
}
```

- @PostConstruct & @PreDestroy
  - @PostConstruct와 @PreDestroy는 각각 Instantiation callback, Destruction callback 메소드를 지정하기 위해 사용한다.

```
public class CachingMovieLister {

   @PostConstruct
   public void populateMovieCache() {
      // populates the movie cache upon initialization...
   }

   @PreDestroy
   public void clearMovieCache() {
      // clears the movie cache upon destruction...
   }
}
```

❑ Annotation Auto Detection

- Spring은 Streotype Annotation 을 사용하여 Bean에 대한 정의를 할 수 있다.
- @Component 를 사용하여 Bean을 정의 할 수 있지만 @Repository, @Service, @Controller annotation을 사용하여, 각각 Persistence, Service, Presentation 레이어의 컴포넌트로 지정하여 특별한 관리 기능을 제공하고 있다. @Repository, @Service, @Controller는 @Component annotation을 상속받고 있다.
- JSR-330에서 제공하는 @Named Annotation을 사용하는 경우 스프링은 @Component와 마찬가지로 Bean으로 관리되도록 지원한다. (단, @Named 를 사용하는 경우 아직은 Scope이 Singleton으로 고정됨)

```
<?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
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="org.example"/>

</beans>
```

- <context:component-scan/> element의 'base-package' attribute는 컴포넌트를 찾을 기본 package이다. ','를 사용하여 다수의 기본 package를 지정할 수 있다.
- <context:component-scan /> 사용하는 경우 기본적으로 <context:annotation-config /> 가 인식되므로 별도로 설정을 안해도 된다.

- 이름 설정

  @Component, @Repository, @Service, @Controller annotation의 name 값으로 bean의 이름을 지정할 수 있다. 아래 예제의 Bean 이름은 "myMovieLister"이다.

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

  - 만약 name 값을 지정하지 않으면, class 이름의 첫문자를 소문자로 변환하여 Bean 이름을 자동으로 생성한다. 아래 예제의 Bean 이름은 "movieFinderImpl"이다.

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

❑ Annotation Auto Detection

- Scope 설정
  - @Scope annotation을 사용하여, 자동으로 찾은 Bean의 scope를 설정할 수 있다.

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

- Qualifier 설정
  - @Qualifier annotation을 사용하여, 자동으로 찾은 Bean의 qualifier를 설정할 수 있다.

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

- Filter를 사용한 Scanning
  - <context:component-scan/> 의 하위 엘리먼트를 사용하여 Bean으로 등록되는 클래스들을 filtering할 수 있다. 필터링시 @Component 를 exclude 하면 하위 어노테이션이 전부 배제됨.

Filter Type	Example	Description
annotation	org.example.SomeAnnotation	type level에 정의된 해당 컴포넌트 annotation 정의
assignable	org.example.SomeClass	해당 컴포턴트로 정의되(상속, 구현) 클래스
aspectj	org.example..*Service+	AspectJ 표현식과 매치되는 컴포넌트
regex	org₩.example₩.Default.*	정규표현식에 매치되는 컴포넌트 클래스 이름
custom	org.example.MyTypeFilter	org.springframework.core.type .TypeFilter interface를 구현한 구현체

```
<context:component-scan base-package="org.example">
    <context:include-filter type="regex" expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation"
                    expression="org.springframework.stereotype.Repository"/>
 </context:component-scan>
```

❑ Bean Definition Profiles

- Spring 3 부터 추가되어 동일한 id의 빈을 여러개 정의하고 사용자의 설정으로 활성화 시킨 profile 에 해당하는 빈이 런타임에 동작하도록 하는 기능

```
<beans profile="dev">
    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
<beans profile="production">
    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>`
```

- Profile 설정 시에는 반드시 특정 profile 의 활성화가 필요함

```
<servlet>
     <servlet-name>dispatcher</servlet-name>
     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
     <init-param>
         <param-name>spring.profiles.active</param-name>
         <param-value>production</param-value>
     </init-param>
 </servlet>
```

❑ Java based Configuration

- Spring 3 부터 Spring Java Configuration 프로젝트의 주요 특징들을 추가하여 자바 기반의 Confiration 정의가 가능하도록 지원함. Injection 속성 정의시 type 오류가 있으면 컴파일부터 수행되지 않아 Type Safety를 보장하며, Bean 구현체가 Spring 에 의존되지 않고, 순수한 POJO로 구현될 수 있음.
  XML 설정을 자바 설정으로 바꾸면 다음과 같다.

```
<bean id="bar" class="x.y.Bar" />
<bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar" />
</bean>
```

```
@Configuration
@Profile({"dev", "production"})
@PropertySource("classpath:appInfo.properties")
public class SampleConfig {
   @Bean
   public Bar bar(){
        return new Bar();
   }

   @Bean(name="foo")
   @Lazy
   @Scope
   public Foo fooGen(){
         return new Foo(bar());
   }
}
```

- @Configuration : 클래스 레벨에서 정의하는 어노테이션으로 이를 포함하고 있는 클래스는 Bean 메타데이터에 대한 정보를 가지며 Spring Container에 의해 처리되는 Configuration 클래스임을 나타냄.
- @Bean
  - 메소드 레벨로 사용하는 어노테이션으로 특정 빈에 대한 정의를 포함하고 있는 메소드임을 나타냄.
  - @Bean 정의 메소드는 해당 Bean의 인스턴스를 생성하여 전달하는 로직을 포함하고 있어야 하며 Spring Container 는 해당 메소드명을 Bean의 이름으로 등록함.
  - name 속성 : 기본적으로 사용되는 메소드명이 아닌 다른 이름으로 변경할 수 있음.
  - initMethod & destroyMethod 속성 : Bean의 라이프사이클 메소드를 지정할때 사용.
- @Lazy : bean 엘리먼트의 lazy-init 속성과 동일한 설정으로, 싱글턴 빈을 인스턴스화하지 않고, 빈에 대한 주입 요청이 발생할 때 인스턴스를 생성함.
- @Scope
  - 빈의 스코프를 결정함. 사용하지 않는 경우 기본적으로 singleton
  - proxyMode : request, session 등의 스코프 빈의 경우 프록시 객체를 생성하는 방법에 대한 정의.
    기본값은 ScopedProxyMode.NO로 되어있으며, INTERFACES인 경우, JDK 의 프록시 생성 API를 사용하고, TARGET_CLASS 인 경우 cglib 라이브러리를 사용하기 때문에 Spring3 버전까지는 의존성을 형성해줘야 했지만, Spring4 부터 Code Generating API 가 Spring 기본 모듈에 포함되어있음.
- @DependsOn : 해당 빈이 초기화되기 전에 먼저 초기화해야되는 빈들의 목록을 설정.
- @Primary : Type Injection을 하는 경우, 대상이 되는 동일 타입의 빈이 여러개가 등록되어있을때 그중 우선 주입될 빈을 설정하기 위한 어노테이션
- @Import, @ImportResource : 다른 Java Configuration이나 Configuration XML에 정의된 빈 정보를 참조하기 위한 어노테이션
- @Value : SpEL을 사용한 value injection을 위한 어노테이션
- @ComponentScan : <context:component-scan>과 동일
- @EnableTransactionManagement : 어노테이션 기반의 트랜잭션 관리 기능을 활성화
- @EnableWebMvc : <mvc:annotation-driven >과 유사하게 Spring MVC 설정에 필요한 전략빈들에 대한 설정을 자동화해줌.
- @Profile : 빈 메타데이터 profile 설정을 위한 어노테이션, Configuration 클래스가 아닌 빈단위의 profile 설정도 가능함. 빈들의 profile을 구성하는 경우, 특정 profile을 활성화시키지 않으면 exception 이 발생함.
- @PropertySource : 외부 설정 파일을 읽어들여 PropertySource 를 구성하기 위한 어노테이션
- 어노테이션 지향적으로 빈 메타데이터에 대한 설정을 마친 후, Spring Container 를 초기화.

```
AnnoationConfigApplicationContext ctx =
              new AnnotationConfigApplicationContext();
// java configruation에서 profile 활성화
ctx.getEnvironment().setActiveProfiles("dev");
ctx.scan("x.y");
ctx.refresh();
```

- Java Configuration 에서 유의할 사항
  - @Configuration 클래스는 기본적으로 빈 캐시를 위해 서브클래스를 생성하게 되는데, 이때 CGLIB 라이브러리를 이용해 생성하므로, Spring3 버전까지는 해당 라이브러리 의존성이 필요함.
  - @Configuration 클래스의 서브클래스를 생성하고, 인스턴스화하여 빈 캐싱과 캐시 체크를 하게 되므로, 반드시 기본 생성자를 가져야함.
  - 서브클래스가 생성되어야 하므로, final 클래스는 configuration 클래스가 될수없음.

❑ ApplicationContext for web application

- Spring은 Web Application에서 ApplicationContext를 쉽게 사용할 수 있도록 각종 class들을 제공하고 있다.
- Servlet 2.4 이상
  - Servlet 2.4 specification부터 Listener를 사용할 수 있다. Listener를 사용하여 ApplicationContext를 설정하기 위해서 web.xml 파일에 다음 설정을 추가한다.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-
value>
</context-param>
<!- Java Configuration 을 사용하는 경우의 설정 위치 지정 -->
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
     org.springframework.web.context.support.AnnotationConfigWebApplicationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>x.y.SampleWebConfig</param-value>
</context-param>

<listener>
  <listener-class>
      org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

  - contextParam의 'contextConfigLocation' 값은 Spring XML 설정 파일의 위치를 나타낸다.

- Servlet 2.3 이하
  - Servlet 2.3이하에서는 Listener를 사용할 수 없으므로, Servlet를 사용한다. web.xml 파일에 다음 설정을 추가

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-
value>
</context-param>

<servlet>
   <servlet-name>context</servlet-name>
   <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-
class>
   <load-on-startup>1</load-on-startup>
</servlet>
```