

## 개요

프레임워크의 기본적인 기능인 Inversion of Control(IoC) Container 기능을 제공하는 서비스이다. 객체의 생성 시, 객체가 참조하고 있는 타 객체에 대한 종속성을 소스 코드 내에서 하드 코딩하는 것이 아닌, 소스 코드 외부에서 설정하게 함으로써, 유연성 및 확장성을 향상시킨다.



- IoC Container
  - 개요
    - 주요 개념
    - 사용된 오픈 소스
  - 설명
    - IoC Container of Spring Framework
  - 참고자료

## 주요 개념

### Inversion of Control(IoC)

IoC는 Inversion of Control의 약자이다. 우리나라 말로 직역해 보면 "역제어"라고 할 수 있다. 제어의 역전 현상이 무엇인지 살펴본다. 기존에 자바 기반으로 어플리케이션을 개발할 때 자바 객체를 생성하고 서로간의 의존 관계를 연결시키는 작업에 대한 제어권은 보통 개발되는 어플리케이션에 있었다. 그러나 Servlet, EJB 등을 사용하는 경우 Servlet Container, EJB Container에게 제어권이 넘어가서 객체의 생명주기(Life Cycle)를 Container들이 전담하게 된다. 이처럼 IoC에서 이야기하는 제어권의 역전이란 객체의 생성에서부터 생명주기의 관리까지 모든 객체에 대한 제어권이 바뀌었다는 것을 의미한다.





### 관련문서

-  Martin Fowler가 저술한  Inversion of Control
- [Inversion of Control 한글 번역](#)

### Dependency Injection

각 클래스 사이의 의존관계를 빈 설정(Beans Definition)정보를 바탕으로 컨테이너가 자동적으로 연결해주는 것을 말한다. 컨테이너가 의존관계를 자동적으로 연결시켜주기 때문에 개발자들이 컨테이너 API를 이용하여 의존관계에 관여할 필요가 없게 되므로 컨테이너 API에 종속되는 것을 줄일 수 있다. 개발자들은 단지 빈 설정파일(저장소 관리 파일)에서 의존관계가 필요하다는 정보를 추가하기만 하면 된다.


### 관련문서

-  Martin Fowler가 저술한  Inversion of Control Containers and the Dependency Injection pattern
-  Inversion of Control Containers and the Dependency Injection pattern 한글 번역 by  최범균(Blog 자바캔 Java Can Do It)

## 사용된 오픈 소스

- IoC Container는  Spring Framework를 수정없이 사용한다.

## 설명

본 IoC Container는 Spring Framework의 기능을 수정없이 사용하는 것으로, 본 가이드 문서는  The Spring Framework - Reference Documentation를 번역 및 요약한 것이다. Spring Framework IoC Container에 대한 상세한 설명이 필요한 경우 The Spring Framework - Reference Documentation 원본 문서 및 Spring Framework API를 참조한다.

## IoC Container of Spring Framework

org.springframework.beans과 org.springframework.context 패키지는 Spring Framework의 IoC Container의 기반을 제공한다. BeanFactory 인터페이스는 객체를 관리하기 위한 보다 진보된 설정 메커니즘을 제공한다. BeanFactory 인터페이스를 기반으로 작성된 ApplicationContext 인터페이스(BeansFactory 인터페이스의 sub-interface이다)는 BeanFactory가 제공하는 기능 외에 Spring AOP, 메시지 리소스 처리(국제화에서 사용됨), 이벤트 전파, 웹 어플리케이션을 위한 WebApplicationContext 등 어플리케이션 레이어에 특화된 context 등의 기능을 제공한다.

요약하면, BeanFactory는 프레임워크와 기본적인 기능에 대한 설정 기능을 제공하는 반면에, ApplicationContext는 좀더 Enterprise 환경에 맞는 기능들을 추가로 제공한다. ApplicationContext는 BeanFactory의 완전한 superset이므로, BeanFactory의 기능 및 행동에 대한 설명은 ApplicationContext에도 모두 해당된다.

본 문서는 크게 두 부분으로 나뉘어지는데, 첫번째 부분은 BeanFactory와 ApplicationContext 모두에 적용되는 기본적인 원리를 설명하고, 두번째 부분은 ApplicationContext에만 적용되는 특징들을 설명한다.

- **Basics**  
IoC Container를 설명하기 위해 필요한 기본적인 개념 및 사용 방법을 설명한다.
- **Dependencies**  
IoC Container의 핵심 기능인 Dependency Injection의 사용 방식 및 설정 방법을 설명한다.
- **Bean scope**  
IoC Container에 의해 관리되는 Bean의 생성 방식 및 적용 범위를 설명한다.
- **Customizing the nature of a bean**  
Bean의 생명주기 관리, Bean이 속한 Container 참조 등 Bean의 성질을 변화시키는 방법을 설명한다.
- **Bean definition inheritance**  
Bean 정의 상속에 대해서 설명한다.
- **Container extension points**  
IoC Container의 기능을 확장하는 방법을 설명한다.
- **The ApplicationContext**  
ApplicationContext만이 제공하는 기능을 설명한다.
- **Annotation-based configuration**  
Java Annotation을 기반으로 Bean을 정의하는 방법을 설명한다.
- **Classpath scanning for managed components**  
Dependency Injection에 의해 삽입되는 base Bean에 대한 Java Annotation 기반 설정 방법을 설명한다.

## 참고자료

- 🌐 The Spring Framework - Reference Documentation / Chapter 3. The IoC container
- 🌐 Inversion of Control
- 🌐 Inversion of Control Containers and the Dependency Injection pattern
- 🌐 Inversion of Control Containers and the Dependency Injection pattern 한글 번역 by 🌐 최범균(Blog 자바캔(Java Can Do It))

## 개요

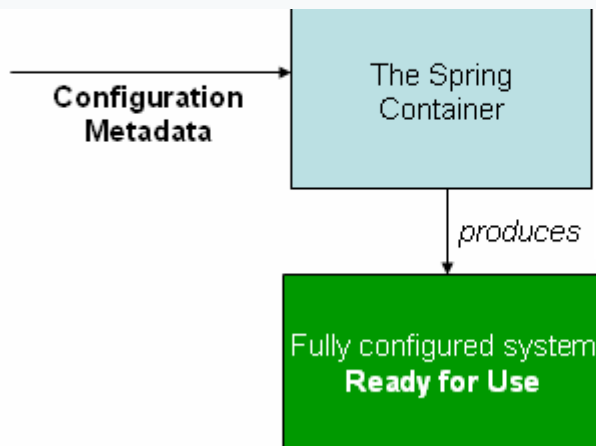
## 설명

- Basics
  - 개요
  - 설명
    - The container
    - The beans
  - 참고자료

Spring Framework에서 Bean은 어플리케이션을 구성하고, IoC container에 의해 관리되어 지는 객체를 의미한다. Bean은 간단히 말해 IoC container에 의해 객체화되고, 조립되고, 또는 관리되는 객체이다. Bean들과 Bean들간의 종속성은 container가 사용하는 설정 메타데이터에 의해 결정된다.

## The container

org.springframework.beans.factory.BeanFactory 인터페이스는 Spring IoC Container의 핵심 인터페이스이다. Spring IoC Container는 객체를 생성하고, 객체간의 종속성을 이어주는 역할을 한다.



## 설정 정보(Configuration Metadata)

위 그림에서 보듯이, Spring IoC container는 설정 정보(configuration metadata)를 필요로한다. 이 설정 정보는 Spring IoC container가 "객체를 생성하고, 객체간의 종속성을 이어줄 수 있도록" 필요한 정보를 제공한다. 설정 정보는 일반적으로 XML 형태로 작성된다. 설정 정보는 XML 형태가 아닌 Java Annotation을 이용하여 설정이 가능하다. Annotation을 사용한 설정 방법은 [Annotation-based configuration](#)에서 설명하고 있다.

아래 예제는 XML 형태의 설정 정보의 기본적인 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>

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

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

</beans>
    
```

<beans> tag는 Spring IoC container의 설정 정보를 나타내는 tag이다. 그리고 각각의 <bean> tag는 Spring IoC container가 생성하고, 관리할 객체의 정의를 나타낸다.

XML 설정 정보를 여러 개의 파일로 나뉘어 구성될 수 있다. 이 경우, 전체 설정 정보를 읽기 위해서 하나의 설정 파일에서 다른 파일을 import할 수 있다. Import 하는 방법으로 <import> tag를 사용한다.

```

<beans>

    <import resource="services.xml" />
    <import resource="resources/messageSource.xml" />
    
```

```

<import resource="/resources/themeSource.xml" />

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

</beans>

```

<import> tag의 resource attribute는 import할 XML 설정 파일의 위치를 나타낸다.

## Container 객체화

다음은 Container를 객체화하는 한 예이다.

```

ApplicationContext context = new ClassPathXmlApplicationContext(
    new String[] { "services.xml", "daos.xml" });

// an Application is also a BeanFactory (via inheritance)
BeanFactory factory = context;

```

위 예제의 ClassPathXmlApplicationContext는 ApplicationContext의 한 종류이며, ApplicationContext 인터페이스는 BeanFactory 인터페이스를 상속하고 있다.

## Container 사용 방법

Container를 객체화하면 getBean(String) 메소드를 사용하여 bean을 가져올 수 있다.

### The beans

Spring IoC container는 다수의 bean들을 관리한다. Container는 설정 정보를 사용하여 bean들은 생성한다. Container에서 사용하는 bean 정의는 아래 정보를 담고 있다.

- 클래스 이름(a package-qualified class name): bean의 실제 구현 클래스를 나타낸다.
- Bean 행동 정보(bean behavioral configuration elements): container 안에서 bean이 어떤 식으로 행동하는지에 대한 정보를 나타낸다.(scope, lifecycle callbacks 등등)
- 다른 bean에 대한 참조(references to other beans): bean이 동작하기 위해 필요한 다른 bean들에 대한 참조 정보를 나타낸다. 이런 참조는 협력자(collaborators) 또는 종속성(dependencies)라고도 한다.
- 기타 객체에 설정할 정보들(other configuration settings): connection pool을 관리하는 bean에서 사용할 connection의 개수, 또는 pool의 최대 크기 등

위 개념적인 정보들은 실제 <bean> tag로 작성된다. <bean> tag를 구성하는 bean 정의는 아래 표와 같다.

Feature	Explained in...
class	Bean 객체화(Instantiation beans)
name	Bean 이름(Naming beans)
scope	Bean scope
constructor arguments	종속성 삽입(Injecting dependencies)
properties	종속성 삽입(Injecting dependencies)
autowiring mode	자동연기(Autowiring collaborators)
dependency checking mode	종속성 검사(Checking for dependencies)
lazy-initialization mode	늦은 객체화(Lazily-instantiated beans)
initialization method	객체화 callbacks(Initialization callbacks)
destruction method	파괴 callbacks(Destruction callbacks)

## Bean 이름(Naming beans)

모든 bean은 하나 이상의 id를 가져야 하며, 각각의 id는 container안에서 단 하나만 존재해야 한다. 일반적으로 대부분의 bean은 하나의 id를 가지지만, 별명(alias)를 사용하여 둘 이상의 id를 가질 수도 있다.

Bean id에 대한 명명 규칙은 Java의 class field 명명 규칙과 같다. id는 소문자로 시작하고, 두번째 단어부터는 첫글자는 대문자로 작성한다. 'accountManager', 'accountService', 'userDao', 'loginController' 등

### Bean 별명(Aliasing beans)

<alias> tag를 사용하여 이미 정의된 bean에게 추가적인 이름을 부여할 수 있다.

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

name attribute는 대상이 되는 bean의 이름이고, alias attribute는 부여할 새로운 이름이다.

## Bean 객체화(Instantiation beans)

모든 bean 정의는 객체화를 위해 실제 Java Class가 필요하다.

XML 설정에서는 'class' attribute를 통해 Java Class를 설정한다. 대부분의 경우 Container는 bean를 객체화하기 위해서 Java의 'new' 연산자를 사용한다. 또는 특수한 경우, static 메소드를 사용할 수도 있다. 본 문서는 생성자를 이용한 객체화만을 설명한다.

생성자를 이용한 객체화는 가장 일반적인 방식으로, 다음과 같이 사용한다.

```
<bean id="exampleBean" class="example.ExampleBean" />
<bean name="anotherExample" class="examples.ExampleBeanTwo" />
```

## 참고자료

-  Spring Framework - Reference Document / 3.2. Basics - containers and beans

## 개요

- Inversion of Control
  - 개요
  - 설명
  - 참고자료

본 문서는 🌍Martin Fowler가 저술한 🌍Inversion of Control 문서를 번역 및 일부 의역한 것이다.

## 설명

Inversion of Control(IoC)는 당신이 프레임워크를 확장할 때 마주치게 되는 일반적인 사상이다. 또한, 프레임워크를 정의하는 특징이기도 하다.

간단한 예제를 생각해보자. 명령줄의 질문을 통해 사용자로부터 어떠한 정보를 입력받는 프로그램을 작성한다고 생각해보자. 나는 아마 다음과 같은 것을 작성할 것이다.

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

위 예제에서, 내가 작성한 코드는 제어권을 가지고 있다 : 질문을 언제 할 것인지, 대답은 언제 읽을 것인지, 그리고 그런 결과들을 언제 처리할 것인지 등을 결정하고 있다.

만약 같은 일을 하기 위해서 윈도우 시스템을 사용한다면, 나는 다음과 같이 윈도우를 설정할 것이다.

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

위 두 프로그램은 제어의 흐름에 있어서 큰 차이점을 가지고 있다. 특히 `process_name`과 `process_quest` 메소드가 호출되는 시점에 대한 제어가 다르다. 명령줄 방식을 사용한 프로그램의 경우, 나는 메소드들이 호출되는 시점을 직접 제어했다. 하지만, 윈도우를 사용한 프로그램은 그러지 않았다. 대신 나는 윈도우 시스템에게 제어권을 넘겨주었다(`Tk.mainloop` 명령어를 사용하여). 윈도우 시스템은 내가 폼을 생성할 때 만든 결합 정보를 이용하여 내 메소드들을 호출할 시점을 결정한다. 즉, 제어가 역전된 것이다 - 내가 프레임워크를 호출하는 것이 아니라 프레임워크가 나를 호출하는 것이다. 이 사상이 Inversion of Control이다 (또한 Hollywood Principle - "Don't call us, we'll call you"라고도 한다).

프레임워크의 가장 중요한 특징은, 사용자가 프레임워크를 사용하기 위해 만든 메소드들이 사용자 어플리케이션 코드에서 호출되는 것보다 프레임워크에 의해 호출되는 것이 더 종종 일어난다는 것이다. 프레임워크는 어플리케이션의 활동을 조합하고 순차적으로 수행하는 메인 프로그램의 역할을 수행한다. 이러한 제어의 역전이 프레임워크가 확장가능한 뼈대로서의 기능을 수행할 수 있도록 해준다. 사용자는 프레임워크가 정의한 일반적인 알고리즘을 확장하여 특정 어플리케이션을 위한 메소드를 생성한다.

🌍--Ralph Johnson and Brian Foote

Inversion of Control은 프레임워크를 라이브러리와 구별짓게 만드는 핵심이다. 라이브러리는 본질적으로 당신이 호출할 수 있는 기능들의 집합이다(요즘은 이러한 기능들이 클래스를 구성하고 있다). 한번 호출되면 작업을 수행하고 클라이언트에게 다시 제어권을 넘긴다.

프레임워크는 일부 추상적인 설계를 가지고 있으며, 미리 정의된 행동 방식을 가지고 있다. 프레임워크를 사용하기 위해서 당신은 프레임워크가 제공하는 클래스를 상속하거나 또는 작성한 클래스를 프레임워크에 삽입함으로써 프레임워크에 존재하는 확장 지점에 당신이 원하는 행동 방식을 삽입해야 한다. 그러면 프레임워크는 각각의 확장 지점에서 당신의 코드를 호출할 것이다.

당신이 만든 코드를 삽입하는 방식에는 여러가지가 있다. 위 ruby 예제의 경우, 우리는 이벤트 이름과 🌍Closure를 변수로 갖는 text entry field의 bind 메소드를 호출했다. Text entry box는 이벤트를 감지할 때 마다 closure의 코드를 호출한다. 이처럼 closure를 이용하는 것은 매우 간편하지만, 이를 지원하는 언어는 많지 않다.

또다른 방법으로는 프레임워크가 이벤트를 정의하고, 클라이언트가 이들 이벤트를 받는 방법이 있다. .NET 플랫폼이 이벤트를 선언할 수 있는 언어적 특징을 가진 좋은 예이다. 당신은 delegate를 사용하여 메소드와 이벤트를 연결할 수 있다.

위 방식(실제로는 둘은 같다)은 단순한 경우에는 잘 동작한다. 하지만 때때로 당신은 하나의 확장 지점에서 여러개의 메소드를 접합하기를 원할 수도 있다. 이런 경우 프레임워크는 인터페이스를 정의하고 클라이언트가 이를 구현할 수 있다.

EJB가 이런 inversion of control 형식의 좋은 예이다. 당신이 session bean을 개발할 때, 당신은 여러 생명주기 지점에서 EJB

테이너에 의해 호출되는 다양한 메소드를 구현할 수 있다. 예를 들어, Session Bean 인터페이스는 `ejbRemote`, `ejbPassivate`(2차 저장소에 저장됨), 그리고 `ejbActivate`(비활성 상태에서 복원됨)를 정의한다. 당신은 이 메소드들이 호출되는 시점에 대한 제어권을 가지지 않고, 다만 무엇을 할 것인지만 결정한다. 컨테이너가 우리를 호출하고, 우리는 그러지 않는다.

Inversion of Control의 복잡한 경우가 존재하지만, 당신은 좀더 단순한 상황에서 효과를 볼 수 있다. Template method가 좋은 예이다: 부모클래스는 제어의 흐름을 정의하고, 자식클래스는 메소드를 재정의하거나 추상메소드를 구현함으로써 확장할 수 있다. JUnit에서, 프레임워크는 당신이 테스트 기반을 생성하고 삭제하기 위해 작성한 `setUp`과 `tearDown` 메소드를 호출한다. 프레임워크가 호출하면, 당신의 코드는 반응한다. 즉, 제어가 역전된 것이다.

요즘 IoC 컨테이너의 등장에 따라 inversion of control의 의미에 대한 일부 혼동이 발생하고 있다. 몇몇 사람들은 이 문서에서 설명한 일반적인 원리와 이들 컨테이너에서 사용하고 있는 inversion of control의 특수한 형식인 dependency injection을 혼동하고 있다. 이름에서 약간 혼란이 야기된다고 할 수 있는데,(또한 모순적이기도 하다) IoC 컨테이너는 일반적으로 EJB의 경쟁상대로 간주되지만 EJB가 inversion of control을 더 많이 사용하기 때문이다.

어원: 내가 알기로, Inversion of Control이란 단어는 1988년 Object-Oriented Programming 저널에 발표된 Johnson and Foote의 논문 [Designing Reusable Classes](#)에서 처음 사용되었다. 이 논문은 잘 작성된 논문 중의 하나로, 발표 이후 15년에 이른 현재까지도 읽을만한 가치가 있다. 그들은 어딘가 다른 곳에서 단어를 가져왔다고 하지만, 어디였는지는 기억하지 못한다. 이 단어는 object-oriented 커뮤니티 속으로 점점 더 녹아들었고 결국 책 Gang of Four에도 나타나게 되었다. 좀 더 화려한 별칭인 'Hollywood Principle'은 1983년 Mesa에 실린 [Richard Sweet](#)의 논문에서 고안된 걸로 보인다. 설계 목표에서 그는 다음과 같이 기술하고 있다. *"Don't call us, we'll call you (Hollywood's Law): A tool should arrange for Tajo to notify it when the user wishes to communicate some event to the tool, rather than adopt an 'ask the user for a command and execute it' model."* John Vlissides는 [column for C++ report](#)에서 'Hollywood Principle'의 개념을 잘 설명하고 있다. (어원에 대해서 도움을 준 Brian Foote과 Ralph Johnson에게 감사드린다.)

## 참고자료

- <http://martinfowler.com/bliki/InversionOfControl.html>

개요	<ul style="list-style-type: none"><li>▪ Dependencies<ul style="list-style-type: none"><li>▪ 개요</li><li>▪ 설명<ul style="list-style-type: none"><li>▪ 종속성 삽입(Injecting dependencies)</li><li>▪ 종속성 상세 설정(Dependencies and configuration in detail)</li><li>▪ depends-on 사용(Using depends-on)</li><li>▪ 늦은 객체화(Lazily-instantiated beans)</li><li>▪ 자동연기(Autowiring collaborators)</li><li>▪ 종속성 검사(Checking for dependencies)</li><li>▪ 메소드 삽입(Method Injection)</li></ul></li><li>▪ 참고자료</li></ul></li></ul>
설명	
종속성 삽입(Injecting dependencies)	

종속성 삽입(Dependency Injection(DI))의 기본적인 원칙은 객체는 단지 생성자나 set 메소드를 통해서만 종속성(필요로 하는 객체)를 정의한다는 것이다. 그러면 container는 bean 객체를 생성할 때, bean이 정의한 종속성을 삽입한다. 이는 Bean이 스스로 필요한 객체를 생성하거나 찾는 등의 제어를 가지는 것과는 반대의 개념으로 Inversion of Control(IoC)라고 부른다.

종속성 삽입에는 두가지 방법이 있다. [Constructor Injection](#)과 [Setter Injection](#)이다.

Constructor Injection

생성자(Constructor) 기반의 DI는 다수의 arguments를 갖는 생성자를 호출하여 종속성을 주입한다. <constructor-arg> element를 사용한다.

```
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>
```

만약, <value>true</value>와 같이 type이 명확하지 않은 값을 사용하는 경우, Spring은 생성자의 어떤 argument에 해당 하는지 결정할 수 없다.

```
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;
    }
}
```

Constructor Argument Type Matching

위와 같은 경우, 'type' attribute를 통해서 각 argument의 타입을 지정할 수 있다.

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

Constructor Argument Index

위와 같은 경우 'index' attribute를 통해서 각 argument의 위치를 지정할 수 있다.

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



(\* index는 0부터 시작한다.)

## Setter Injection

Setter기반의 DI는 argument가 없는 생성자를 통해 bean 객체가 생성된 후, setter 메소드를 호출하여 종속성을 주입, <property> element를 사용한다.

```
<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"/>
```

```
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;
    }
}
```

## 종속성 상세 설정(Dependencies and configuration in detail)

본 장은 종속성 삽입에 사용되는 <constructor-arg>와 <property> element의 sub-element type을 설명한다.

### 명확한 값(Straight values(primitives, Strings, etc.))

사람이 인식 가능한 문자열 형태를 <value> tag를 사용하여 표현한다. String을 argument나 property의 type에 맞춰 변환해준다.

```
<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>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
  <property name="password">
    <value>masterkaoli</value>
  </property>
</bean>
```

<value> element 대신 'value' attribute를 사용할 수도 있다.

```
<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"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</bean>
```

### 다른 bean 참조(References to other beans(collaborators))

ref element는 container 안에 있는 다른 bean을 참조한다. 참조할 객체를 지정하는 방식에는 3가지가 있다.

#### 1. bean attribute

가장 일반적인 형태로 같은 container 또는 부모 container에 포함된 bean 객체를 참조한다. 'bean' attribute는 대상 bean의 'id' 또는 여러 'name'들 중 하나와 같아야 한다.

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

## 2. local attribute

같은 XML 설정 파일 내의 bean 객체를 참조한다. 'local' attribute는 반드시 대상 bean의 'id'와 같아야 한다. 만약 대상 bean이 같은 XML 파일에 존재한다면 local을 사용하는 것이 좋다.

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

## 3. parent attribute

현재 container의 부모 container의 bean 객체를 참조한다. 'parent' attribute는 대상 bean의 'id' 또는 여러 'name'들 중 하나와 같아야 한다.

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

```
<!-- in the child (descendant) context -->
<bean id="accountService" <-- notice that the name of this bean is the same as the name of the 'parent'
bean
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService" /> <-- notice how we refer to the parent bean
    </property>
    <!-- insert other configuration and dependencies as required as here -->
</bean>
```

## Inner beans

<property/> 또는 <constructor-arg/> element 안에 있는 <bean/> element를 *inner bean*이라고 한다. Inner bean은 id나 name을 정의할 필요가 없다. 정의한다 해도 container에서 무시하기 때문에 정의하지 않는 것이 좋다.

```
<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>
```

Inner bean의 'scope' flag와 'id', 'name'은 무시된다. Inner bean의 scope은 항상 prototype이다. 따라서 inner bean을 다른 bean에 주입하는 것은 불가능하다.

## Collections

Java Collection 타입인 List, Set, Map, Properties를 표현하기 위해 <list/>, <set/>, <map/>, <props/> element가 사용된다.

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- 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>
    <!-- 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>
</bean>
```

```

        </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>
</bean>

```

map의 key와 value, set의 value의 값은 아래 element 중 하나가 될 수 있다.

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

## Collection 병합(Collection merging)

Container는 collection 병합 기능을 제공한다. Bean 정의 상속을 사용하여 부모 bean 정의의 <list/>, <map/>, <set/>, <props/> element와 자식 bean 정의의 <list/>, <map/>, <set/>, <props/> element를 병합할 수 있다.

```

<beans>
<bean id="parent" abstract="true" class="example.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
<bean id="child" parent="parent">
    <property name="adminEmails">
        <!-- the merge is specified on the *child* collection definition -->
        <props merge="true">
            <prop key="sales">sales@example.com</prop>
            <prop key="support">support@example.co.uk</prop>
        </props>
    </property>
</bean>
</beans>

```

위 설정에 따라 생성된 child bean 객체의 adminEmails는 아래와 같은 값을 가진다.

```

administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk

```

## Nulls

null 값을 사용하기 위해서 <null/> element를 사용한다. Spring은 argument가 없을 경우 빈 문자열("")로 인식한다.

```

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

```

위 설정에 따르면, email의 값은 ""이다. 다음은 null 값을 갖는 예제이다.

```

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

```

## 간편한 설정 방법(Shortcuts and other convenience options for XML-based configuration metadata)

### XML-based configuration metadata shortcuts

<property/>, <constructor-arg/>, <entry/> element는 모두 <value/> element 대신에 'value' attribute를 사용할 수 있다.

```

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

```

```

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

```

```

<entry key="myKey">
    <value>hello</value>
</entry>

```

위 설정은 아래와 동일한 설정이다.

```

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

```

```

<constructor-arg value="hello"/>

```

```
<entry key="myKey" value="hello"/>
```

<property/>, <constructor-arg/> element는 <ref/> element 대신에 'ref' attribute를 사용할 수 있다.

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

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

위 설정은 아래와 동일한 설정이다.

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

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

단, shortcut은 <ref bean="xxx">와 동일하다. <ref local="xxx">에 해당하는 shortcut은 없다.

<entry/> element는 'key' / 'key-ref'와 'value' / 'value-ref' attribute를 사용할 수 있다.

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

위 설정은 아래와 같은 설정이다.

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

## The p-namespace and how to use it to configure properties

"<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-2.5.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>
```

아래 예제는 다른 bean 객체의 참조를 삽입하는 예제이다. Attribute 이름 끝에 '-ref'를 붙이면 참조로 인식한다.

```
<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-2.5.xsd">

  <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>
</beans>
```

## Compound property names

복합 형식의 property 이름도 사용 가능하다.

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

foo bean은 fred property를 가지고, fred property는 bob property를 가진다. 그리고 bob property는 sammy property를 가지고, 마지막 sammy property가 123을 값으로 가진다. 이 작업이 정상적으로 동작하려면 bean이 생성되었을 때, foo의

fred property, fred의 bob property는 반드시 null이 아니어야 한다. 그렇지 않을 경우 NullPointerException이 발생한다.

## depends-on 사용(Using depends-on)

대부분의 경우, bean들간의 종속성은 "<ref/>" element에 의해 표현된다. 하지만 드물게 이런 종속성이 직접 나타나지 않는 경우도 있다(예를 들면, database driver 등록처럼 static 메소드에 의해 초기화되어야 하는 경우 등). 이런 경우 'depends-on' attribute를 사용하여 명시적으로 종속성을 표현할 수 있다.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

만약 다수의 bean에 대한 종속성을 표현하고 하는 경우에는, 'depends-on' attribute의 값으로 bean 이름을 나열하면 된다. bean 이름의 구분자로는 콤마(','), 공백문자(' '), 세미콜론(';') 등을 사용할 수 있다.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
  <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

## 늦은 객체화(Lazily-instantiated beans)

ApplicationContext는 시작시에 모든 singleton bean을 선택체화(pre-instantiate)한다. 선택체화(pre-instantiate)는 초기화 과정에서 모든 singleton bean을 생성하고 설정한다는 것을 의미한다.

일반적으로 선택체화가 좋은 방식인데 잘못된 설정이 있는 경우, 즉시 발견할 수 있기 때문이다.

어쨌거나, 이런 방식을 원하지 않을 경우도 있다. 만약 ApplicationContext에 의해 선택체화되는 singleton bean을 원하지 않을 경우, 선택적으로 bean 정의에 늦은 객체화(lazy-initialized)를 설정할 수 있다. 늦은 객체화(lazy-initialized)로 설정된 bean은 시작 시에 생성되는 것이 아니라, 처음으로 필요로 했을때 생성된다.

XML 설정에서는 "<bean/>" element의 'lazy-init' attribute를 사용한다.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean" />
```

늦은 객체화에 대해서 이해하고 있어야 하는 것은, 만약 늦은 객체화로 설정된 bean에 대해서 그렇지 않은 singleton bean이 종속성을 가지고 있다면, ApplicationContext는 시작 시에 singleton bean이 종속하고 있는 모든 bean을 생성한다는 것이다. 즉, 명시적으로 늦은 객체화로 선언한 bean이라도 시작 시에 생성될 수 있다.

그리고, <beans/> element의 'default-lazy-init' attribute를 사용하여 container 레벨에서의 늦은 객체화를 설정할 수 있다.

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

## 자동연기(Autowiring collaborators)

Spring container는 서로 관계된 bean들을 자동으로 엮어(autowire)줄 수 있다. 자동연기(autowiring)는 각각의 bean 단위로 설정된다. 자동연기(autowiring) 기능을 사용하면 property나 생성자 argument를 지정할 필요가 없어지므로, 타이핑일 줄일 수 있다. 자동연기(autowiring)에는 5가지 모드가 있으며, XML 기반 설정에서는 <bean/> element의 'autowire' attribute를 사용하여 설정할 수 있다.

Mode	설명
no	자동연기를 사용하지 않는다. Bean에 대한 참조는 ref element를 사용하여 지정해야만 한다. 이 모드가 기본(default)이다.
byName	Property 이름으로 자동연기를 수행한다. Property의 이름과 같은 이름을 가진 bean을 찾아서 엮어준다.
byType	Property 타입으로 자동연기를 수행한다. Property의 타입과 같은 타입을 가진 bean을 찾아서 엮어준다. 만약 같은 타입을 가진 bean이 container에 둘 이상 존재할 경우 exception이 발생한다. 만약 같은 타입을 가진 bean이 존재하지 않는 경우, 아무 일도 발생하지 않는다; 즉, property에는 설정되지 않는다.
constructor	byType과 유사하지만, 생성자 argument에만 적용된다. 만약 같은 타입의 bean이 존재하지 않거나 둘 이상 존재할 경우, exception이 발생한다.
autodetect	Bean class의 성질에 따라 constructor와 byType 모드 중 하나를 선택한다. 만약 default 생성자가 존재하면, byType 모드가 적용된다.

만약 종속성을 property나 constructor-arg를 사용하여 명시적으로 설정한 경우, 자동연기(autowiring) 설정은 무시된다.

## Bean을 자동역기 대상에서 제외하는 방법(Excluding a bean from being available from autowiring)

<bean/> element의 'autowire-candidate' attribute 값을 'false'로 설정함으로써, 대상 bean이 다른 bean에 의해 자동역임을 당하는 것을 방지할 수 있다.

## 종속성 검사(Checking for dependencies)

Spring IoC container는 bean의 미해결 종속성의 존재를 검사할 수 있다. 이 기능은 bean의 모든 property가 지정되었는지는 확인하고 싶을 때 유용하다. 종속성 검사(Dependency checking) 기능은 자동역기(autowiring) 기능과 마찬가지로 각각의 bean마다 설정할 수 있다. 종속성 검사에는 4가지 모드가 있으며, XML 기반 설정에서는 <bean/> element의 'dependency-check' attribute를 사용하여 설정할 수 있다.

Mode	설명
none	종속성 검사를 하지 않는다. 기본(default) 모드이다.
simple	Primitive 타입과 collection에 대해서 종속성 검사를 수행한다.
object	관련된 객체에 대해서만 종속성 검사를 수행한다.
all	Primitive 타입과 collection, 관련된 객체에 대해서 종속성 검사를 수행한다.

## 메소드 삽입(Method Injection)

대부분의 어플리케이션에서, container에 존재하는 대부분의 bean은 singleton이다. Singleton bean이 다른 singleton bean과 협력(collaborate)하거나, non-singleton bean이 다른 non-singleton bean과 협력하는 경우, 가장 일반적인 방법은 bean의 property를 정의함으로써 종속성을 조절하는 것이다. 하지만 만약 관련된 bean들의 생명주기가 다른 경우 문제가 발생한다. Singleton bean A가 non-singleton bean B를 사용한다고 할 때, container는 singleton bean A를 단지 한번만 생성할 것이고, 따라서 property도 역시 한번만 설정될 것이다. Container는 bean B가 필요한 매 순간 새로운 객체를 생성하여 bean A에게 제공해야 하지만, 그럴 수 있는 방법이 없다.

위 문제에 대한 한가지 해법은 몇몇 제어의 역전(inversion of control)를 버리는 것이다. Bean A는 BeanFactoryAware interface를 구현함으로써 자신이 속한 container를 알 수 있다. 그리고 bean B의 객체가 필요한 순간에 container의 getBean("B")을 호출함으로써 bean B의 객체를 가져올 수 있다.

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// lots of Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;

public class CommandManager implements BeanFactoryAware {

    private BeanFactory beanFactory;

    public Object process(Map commandState) {
        // grab a new instance of the appropriate Command
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // the Command returned here could be an implementation that executes asynchronously, or whatever
    protected Command createCommand() {
        return (Command) this.beanFactory.getBean("command"); // notice the Spring API dependency
    }

    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }
}
```

위 예제는 일반적으로는 바람직하지 않은 솔루션이다. 왜냐하면 업무 코드(business code)는 Spring Framework과 관련될 필요가 없기 때문이다. 메소드 삽입(Method Injection)은 이런 경우를 말끔히 해결할 수 있는 방법이다.

## Lookup 메소드 삽입(Lookup method injection)

Lookup 메소드 삽입은 container가 관리하고 있는 bean의 메소드를 덮어써서(override) container 안에 있는 다른 bean을 찾을 수 있게 하는 기능이다. Spring Framework는 메소드 삽입을 구현하기 위해서 CGLIB 라이브러리를 사용하여 동적으로 상속클래스를 생성한다.

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
    }
}
```

```
// set the state on the (hopefully brand new) Command instance
command.setState(commandState);
return command.execute();
}

// okay... but where is the implementation of this method?
protected abstract Command createCommand();
}
```

삽입될 메소드는 반드시 다음과 같은 형태를 가져야 한다.

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

만약 메소드가 `abstract`이면, 동적으로 생성된 서브클래스는 메소드를 구현할 것이다. 만약 그렇지 않으면 동적으로 생성된 서브클래스를 원본 클래스의 메소드를 덮어쓸(override) 것이다.

```
<!-- 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>
```

`commandManager`는 `command` bean의 새로운 객체가 필요할 때마다 자신의 `createCommand()` 메소드를 호출할 것이다. 만약 `command` bean이 `prototype`이 아닌 `singleton`인 경우, `createCommand` 메소드는 같은 객체를 리턴할 것이다.

동적 서브클래스 생성이 동작하려면 `classpath`에 `CGLIB`가 추가되어 있어야 한다. 그리고 원본 class는 `final`이면 안되며, 덮어쓸(override) 메소드 역시 `final`이면 안된다.

## 참고자료

-  Spring Framework - Reference Document / 3.3. Dependencies

Table of Contents

▪ Bean scope

▪ 개요

▪ 설명

▪ The singleton scope

▪ The prototype scope

▪ Prototype bean에 종속적인 singleton bean(Singleton beans with prototype-bean dependencies)

▪ 기타 scopes(The other scopes)

▪ 참고자료

개요

설명

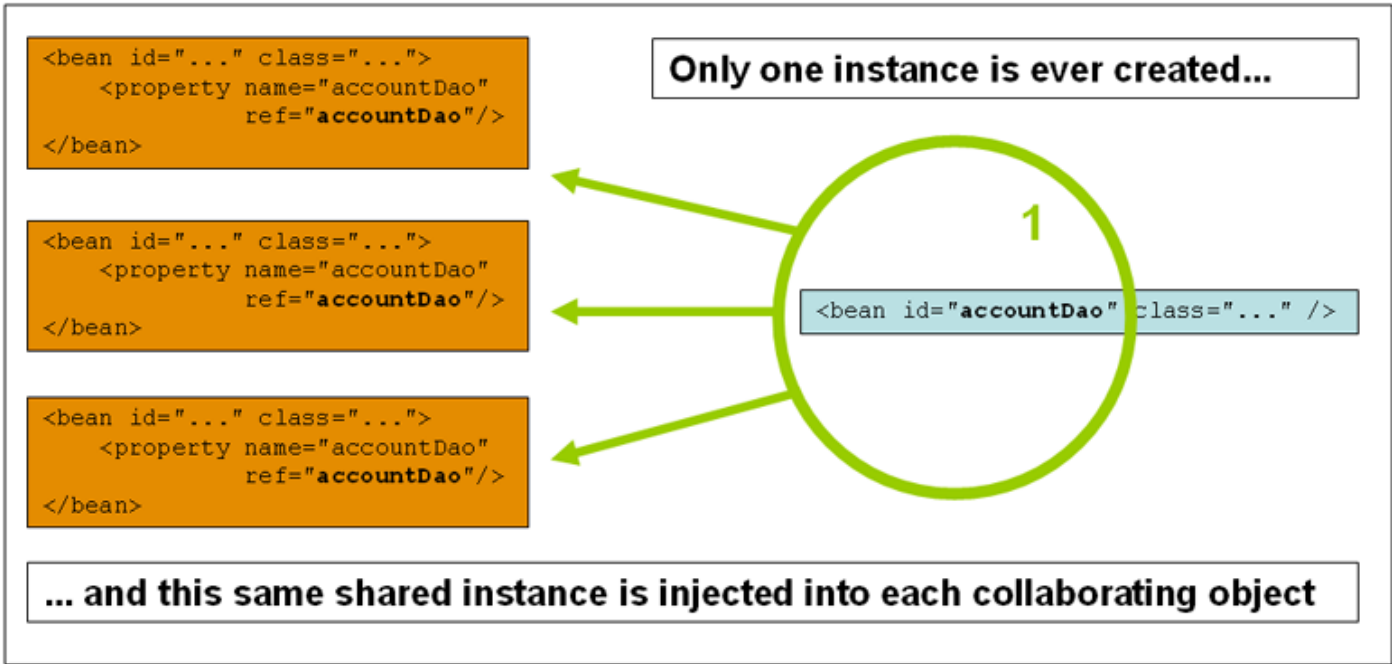
Bean 정의는 실제 bean 객체를 생성하는 방식을 정의하는 것이다. Class와 마찬가지로 하나의 Bean 정의에 해당하는 다수의 객체가 생성될 수 있다

Bean 정의를 통해 객체에 다양한 종속성 및 설정값을 주입할 수 있을 뿐 아니라, 객체의 범위(scope)를 정의할 수 있다. Spring Framework는 5가지 scope을 제공한다(그 중 3가지는 web-aware ApplicationContext를 사용할 때 이용할 수 있다).

Scope	설명
singleton	하나의 Bean 정의에 대해서 Spring IoC Container 내에 단 하나의 객체만 존재한다.
prototype	하나의 Bean 정의에 대해서 다수의 객체가 존재할 수 있다.
request	하나의 Bean 정의에 대해서 하나의 HTTP request의 생명주기 안에 단 하나의 객체만 존재한다; 즉, 각각의 HTTP request는 자신만의 객체를 가진다. Web-aware Spring ApplicationContext 안에서만 유효하다.
session	하나의 Bean 정의에 대해서 하나의 HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. Web-aware Spring ApplicationContext 안에서만 유효하다.
global session	하나의 Bean 정의에 대해서 하나의 global HTTP Session의 생명주기 안에 단 하나의 객체만 존재한다. 일반적으로 portlet context 안에서 유효하다. Web-aware Spring ApplicationContext 안에서만 유효하다.

The singleton scope

Bean이 singleton인 경우, 단지 하나의 공유 객체만 관리된다.



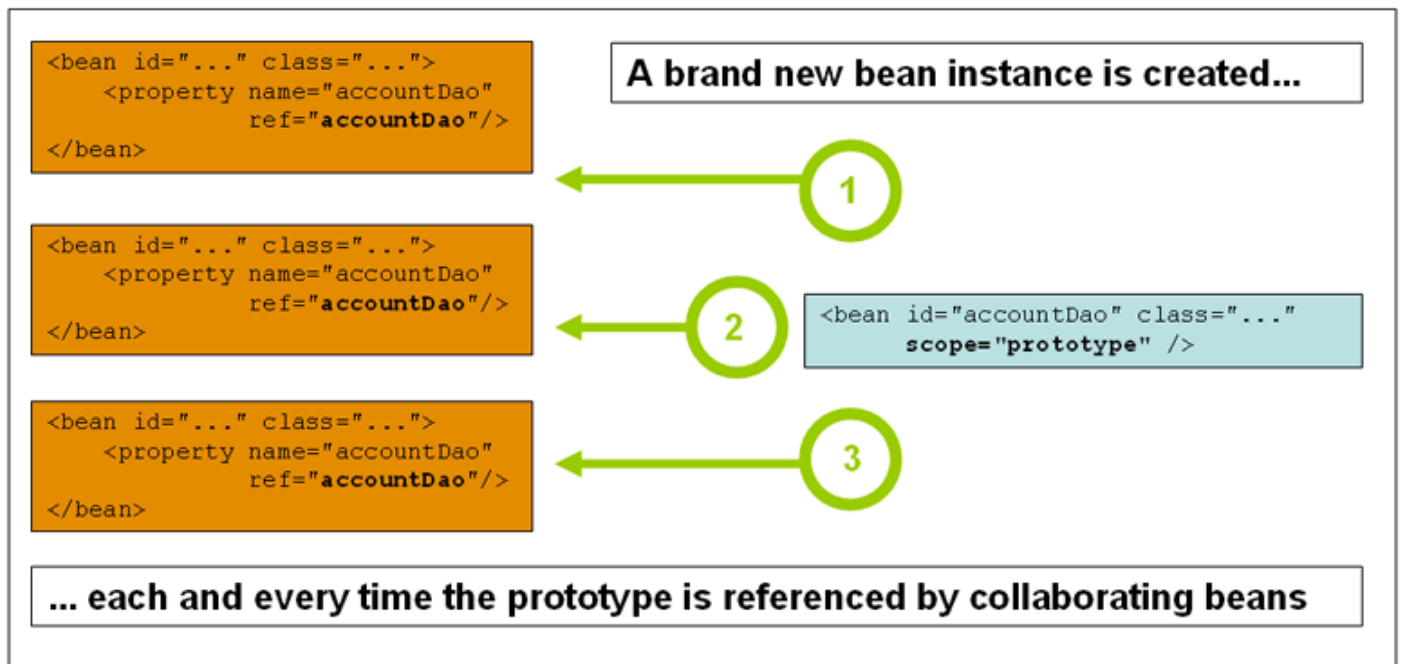
Singleton scope은 Spring의 기본(default) scope이다.

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
<!-- the following is equivalent, though redundant (singleton scope is the default); using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

The prototype scope

Singleton이 아닌 prototype scope의 형태로 정의된 bean은 필요한 매 순간 새로운 bean 객체가 생성된다.





```
<!-- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype" />
<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false" />
```

Prototype scope을 사용할 때 염두에 두고 있어야 할 것이 있다. Spring은 prototype bean의 전체 생명주기를 관리하지 않는다. container는 객체화하고, 값을 설정하고, 다른 prototype 객체와 조립하여 Client에게 전달한 후 더이상 prototype 객체를 관리하지 않는다. 즉, scope에 관계없이 초기화(initialization) 생명주기 callback 메소드가 호출되는 반면에, prototype의 경우 파괴(destruction) 생명주기 callback 메소드는 호출되지 않는다. 이것은 client 코드가 prototype 객체를 clean up하고 prototype 객체가 들고 있던 리소스를 해제하는 책임을 가진다는 것을 의미한다.

## Prototype bean에 종속적인 singleton bean(Singleton beans with prototype-bean dependencies)

이 내용은 메소드용 삽입(Method Injection)에서 다루고 있다.

## 기타 scopes(The other scopes)

request, session, global session scope들은 반드시 web-based 어플리케이션에서 사용할 수 있다.

### 기본 Web 설정(Initial web configuration)

request, session, global session scope을 사용하기 위해서는 추가적인 설정이 필요하다. 추가 설정은 사용할 Servlet 환경에 따라 달라진다.

만약 Spring Web MVC 안에서 bean에 접근할 경우, 즉 Spring DispatcherServlet 또는 DispatcherPortlet에서 처리되는 요청인 경우, 별도의 추가 설정은 필요하다: DispatcherServlet과 DispatcherPortlet은 이미 모든 관련있는 상태를 제공한다.

만약 Servlet 2.4+ web container를 사용하고, JSF나 Struts 등과 같이 Spring의 DispatcherServlet의 외부에서 요청을 처리하는 경우, 다음 javax.servlet.ServletRequestListener를 'web.xml' 파일에 추가해야 한다.

```
<web-app>
...
<listener>
  <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
</listener>
...
</web-app>
```

만약 다른 오래된 web container(Servlet 2.3)를 사용한다면, 제공되는 javax.servlet.Filter 구현체를 사용해야 한다. (filter mapping은 web 어플리케이션 설정에 따라 달라질 수 있으므로, 적절히 수정해야 한다.)

```
<web-app>
...
<filter>
  <filter-name>requestContextFilter</filter-name>
  <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>requestContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>
```

## The 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.

## The 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` 객체를 생성할 것이다. 즉, 'userPreferences' bean은 HTTP Session 수준에 한정된다(scoped). HTTP Session이 폐기될 때, 한정된(scoped) bean로 폐기된다.

## The 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 정의한다. global session scope으로 설정된 bean은 global portlet Session의 일생에 한정된다.

## 한정적 bean에 대한 종속성(Scoped beans as dependencies)

HTTP request 또는 Session에 한정적인(scoped) bean을 정의하는 것은 꽤 괜찮은 기능이지만 Spring IoC Container가 제공하는 핵심 기능은 객체를 생성하는 것 뿐만 아니라 엮어준다는 것이다. 만약 HTTP request에 한정적인(scoped) bean을 다른 bean에 주입하기를 원한다면, 한정적(scoped) bean 대신에 AOP Proxy를 주입해야 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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- a 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>
</beans>
```

Proxy를 생성하기 위해서 `<aop:scoped-proxy/>` element를 scoped bean 정의에 추가해야 한다(CGLIB 라이브러리도 classpath에 추가해야 한다).

## 참고자료

- Spring Framework - Reference Document / 3.4. Bean scopes

개요

설명

- Customizing the nature of a bean
  - 개요
  - 설명
    - Lifecycle callbacks
    - Knowing who you are
  - 참고자료

Lifecycle callbacks

Spring Framework는 container 내부의 bean의 작업을 변화시킬 수 있는 다양한 callback interface를 제공한다.

객체화 callbacks(Initialization callbacks)

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.

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

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

위 예제는 아래 예제와 같다.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

파괴 callbacks(Destruction callbacks)

org.springframework.beans.factory.DisposableBean interface를 구현하면, container가 파괴될 때 bean이 callback를 받을 수 있다. DisposableBean interface는 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 사용한다.

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

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

위 예제는 아래 예제와 같다.

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

## 기본 객체화 및 파괴 메소드(Default initialization & destroy methods)

Spring container는 모든 bean에 대해서 같은 이름의 초기화 및 파괴 메소드를 지정할 수 있다.

```
public class DefaultBlogService implements BlogService {
    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }
}
```

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

`<beans/>` element의 'default-init-method' attribute를 이용하여 기본 객체화 callback 메소드를 지정할 수 있다. 파괴 callback 메소드의 경우 'default-destroy-method' attribute를 이용하여 지정할 수 있다.

`<bean/>` element에 'init-method', 'destroy-method' attribute가 정의되어 있는 경우, 기본값은 무시된다.

## 생명주기 메커니즘 병합

Spring 2.5에서는 3가지 방식의 생명주기 메커니즘이 존재한다. `InitializingBean`과 `DisposableBean` interface, 맞춤 `init()` 과 `destroy()` 메소드, 그리고 [@PostConstruct and @PreDestroy annotations](#) 이다.

만약 서로 다른 생명주기 메커니즘을 같이 사용할 경우, 개발자는 적용되는 순서를 알고 있어야 한다. 객체화 메소드의 순서는 다음과 같다.

1. `@PostConstruct` annotation이 있는 메소드
2. `InitializingBean` callback interface에 정의된 `afterPropertiesSet()`
3. 맞춤 `init()` 메소드

파괴 메소드의 호출 순서는 다음과 같다.

1. `@PreDestroy` annotation이 있는 메소드
2. `DisposableBean` callback interface에 정의된 `destroy()`
3. 맞춤 `destroy()` 메소드

## Knowing who you are

### BeanFactoryAware

`org.springframework.beans.factory.BeanFactoryAware` interface를 구현한 class는 자신을 생성한 `BeanFactory`를 참조할 수 있다.

```
public interface BeanFactoryAware {
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

`BeanFactoryAware` interface를 사용하면, 자신을 생성한 `BeanFactory`를 알 수 있고, 프로그램적으로 다른 bean을 검색할 수 있다. 하지만 이 방법은 Spring과의 결합을 발생시키고, Inversion of Control 스타일에도 맞지 않으므로 피하는 것이 좋다.

대안으로는 `org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean`을 사용할 수 있다.

`ObjectFactoryCreatingFactoryBean`은 `FactoryBean`을 구현한 것으로 bean을 찾아주는 객체를 참조할 수 있다. `ObjectFactoryCreatingFactoryBean`은 스스로 `BeanFactoryAware` interface를 구현하고 있다.

```
package x.y;

public class NewsFeed {
    private String news;
```

```

    public void setNews(String news) {
        this.news = news;
    }

    public String getNews() {
        return this.toString() + ": '" + news + "'";
    }
}

```

```

package x.y;

import org.springframework.beans.factory.ObjectFactory;

public class NewsFeedManager {

    private ObjectFactory factory;

    public void setFactory(ObjectFactory factory) {
        this.factory = factory;
    }

    public void printNews() {
        // here is where the lookup is performed; note that there is no
        // need to hard code the name of the bean that is being looked up...
        NewsFeed news = (NewsFeed) factory.getObject();
        System.out.println(news.getNews());
    }
}

```

아래 설정은 `ObjectFactoryCreatingFactoryBean` 방식을 사용하여 위 두 class를 엮어주는 예제이다.

```

<beans>
  <bean id="newsFeedManager" class="x.y.NewsFeedManager">
    <property name="factory">
      <bean
class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
        <property name="targetBeanName">
          <idref local="newsFeed" />
        </property>
      </bean>
    </property>
  </bean>
  <bean id="newsFeed" class="x.y.NewsFeed" scope="prototype">
    <property name="news" value="... that's fit to print!" />
  </bean>
</beans>

```

## BeanNameAware

`org.springframework.beans.factory.BeanNameAware` interface를 구현한 bean은 container내에서의 *name*을 알 수 있다.

## 참고자료

-  Spring Framework - Reference Document / 3.5. Customizing the nature of a bean

개요

- Bean definition inheritance
  - 개요
  - 설명
  - 참고자료

설명

Bean 정의는 많은 양의 설정 정보를 포함하고 있다. 자식 bean 정의는 부모 bean 정의로부터 설정 정보를 상속받은 bean 정의를 의미한다. 자식 bean 정의는 필요에 따라 부모 bean 정의로부터 상속받은 설정 정보를 덮어쓰거나 추가할 수 있다. XML 기반 설정에서는 자식 bean 정의에 'parent' attribute를 사용하여 상속관계를 정의할 수 있다.

```
<bean id="inheritedTestBean" abstract="true"
      class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from parent -->

</bean>
```

자식 bean 정의는 bean class가 명기되어 있지 않을 경우, 부모 bean 정의의 값을 사용한다. 만약 자식 bean 정의에 bean class가 명기되어 있는 경우, 자식 bean 정의의 bean class는 부모 bean 정의의 모든 property 값을 받아들일 수 있어야 한다.

자식 bean 정의는 부모 bean 정의의 생성자 argument 값, property 값, 그리고 메소드 덮어쓰기를 상속받는다. 만약 init-method, destroy-method, static factory 메소드 설정이 명기되어 있을 경우, 부모의 설정을 덮어쓴다.

다음 설정은 항상 자식 bean 정의의 값을 따른다: *depends on, autowire mode, dependency check, singleton, scope, lazy init*.

부모 bean 정의는 abstract attribute를 사용하여 abstract로 설정할 수 있다. 이 경우, 부모 bean 정의는 class를 지정하지 않는다.

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

부모 bean 정의는 완전하지 않기 때문에 객체화 될 수 없다.

참고자료

-  Spring Framework - Reference Document / 3.6. Bean definition inheritance

개요

설명

Spring Framework의 IoC 컴포넌트는 확장을 고려하여 설계되었다. 일반적으로 어플리케이션 개발자가 다양한 `BeanFactory` 또는 `ApplicationContext` 구현 클래스를 상속받을 필요 없이 "Spring IoC container는 특별한 통합 interface의 구현체를 삽입하여 확장할 수 있다."

- Container extension points
  - 개요
  - 설명
    - BeanPostProcessors를 사용한 확장(Customizing beans using BeanPostProcessors)
    - BeanFactoryPostProcessors를 사용한 확장(Customizing configuration metadata with BeanFactoryPostProcessors)
    - FactoryBeans를 사용한 확장(Customizing instantiation logic using FactoryBeans)
  - 참고자료

BeanPostProcessors를 사용한 확장(Customizing beans using BeanPostProcessors)

`BeanPostProcessors` interface는 다수의 callback 메소드를 정의하고 있는데, 어플리케이션 개발자는 이들 메소드를 구현함으로써 자신만의 객체화 로직(instantiation logic), 종속성 해결 로직(dependency-resolution logic) 등을 제공할 수 있다.

`org.springframework.beans.factory.config.BeanPostProcessor` interface는 두개의 callback 메소드로 구성되어 있다. 특정 class가 Container에 post-processor로 등록되면, post-processor는 container에서 생성되는 각각의 bean 객체에 대해서, container 객체화 메소드 전에 callback을 받는다.

중요한 것은 `BeanFactory`는 post-processor를 다루는 방식에 있어서 `ApplicationContext`와는 조금 다르다. `ApplicationContext`는 `BeanPostProcessor` interface를 구현한 bean을 *자동적으로 인식하고* post-processor로 등록한다. 하지만 `BeanFactory` 구현을 사용하면 post-processor는 다음과 같이 명시적으로 등록되어야 한다.

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);  
// now register any needed BeanPostProcessor instances  
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();  
factory.addBeanPostProcessor(postProcessor);  
  
// now start using the factory
```

명시적인 등록은 불편하기 때문에 대부분의 Spring 기반 어플리케이션에서는 순수 `BeanFactory` 구현보다는 `ApplicationContext` 구현을 사용한다.

Example: Hello World, BeanPostProcessor-style

본 예제는 올바른 예는 아니지만, 기본적인 사용 방법을 보여주기 위함이다.

```
package scripting;  
  
import org.springframework.beans.factory.config.BeanPostProcessor;  
import org.springframework.beans.BeansException;  
  
public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {  
    // simply return the instantiated bean as-is  
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
        return bean; // we could potentially return any object reference here...  
    }  
  
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());  
        return bean;  
    }  
}
```

```
<?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:lang="http://www.springframework.org/schema/lang"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.5.xsd">  
  
    <lang:groovy id="messenger"  
        script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">  
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>  
    </lang:groovy>  
  
    <!--  
        when the above bean ('messenger') is instantiated, this custom  
        BeanPostProcessor implementation will output the fact to the system console  
    -->  
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>  
  
</beans>
```

`InstantiationTracingBeanPostProcessor`는 단순히 정의된다. 비록 이름을 가지고 있지는 않지만 bean이기 때문에 다른 bean과 같이 종속성은 삽입될 수 있다.

## BeanFactoryPostProcessors를 사용한 확장(Customizing configuration metadata with BeanFactoryPostProcessors)

`org.springframework.beans.factory.config.BeanFactoryPostProcessor`는 `BeanPostProcessor`와 의미적으로 비슷하지만, 큰 차이점 중 하나는 `BeanFactoryPostProcessors`는 bean 설정 메타정보를 처리한다는 것이다. Spring IoC container는 `BeanFactoryPostProcessors`가 설정 메타정보를 읽고, container가 실제로 bean를 객체화하기 전에 그 정보를 변경할 수 있도록 허용한다.

Bean factory post-processor는 `BeanFactory`의 경우 수동으로 실행되고, `ApplicationContext`의 경우 자동으로 실행된다.

`BeanFactory`에서는 다음과 같이 수동으로 실행한다.

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));  
  
// bring in some property values from a Properties file  
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();  
cfg.setLocation(new FileSystemResource("jdbc.properties"));  
  
// now actually do the replacement  
cfg.postProcessBeanFactory(factory);
```

### Example: the PropertyPlaceholderConfigurer

`PropertyPlaceholderConfigurer`는 `BeanFactory` 정의로부터 property 값을 분리하기 위해 사용한다. 분리된 값은 Java Properties 형식으로 작성된 다른 파일로 분리된다. 이 방식은 주 XML 설정 파일을 변경하지 않고, 환경 변수 등을 변경될 때 유용하다 (예를 들어 database URLs, 사용자명, 패스워드 등).

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
  <property name="locations">  
    <value>classpath:com/foo/jdbc.properties</value>  
  </property>  
</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}"/>  
  <property name="password" value="{jdbc.password}"/>  
</bean>
```

위 설정의 실제 값은 아래와 같다.

```
jdbc.driverClassName=org.hsqldb.jdbcDriver  
jdbc.url=jdbc:hsqldb:hsqldb://production:9002  
jdbc.username=sa  
jdbc.password=root
```

만약 Spring 2.5부터 지원되는 `context namespace`를 사용하면 다음과 같이 설정할 수 있다.

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

`PropertyPlaceholderConfigurer`는 사용자가 지정한 Properties 파일뿐 아니라 만약 지정한 property가 없을 경우, Java System properties도 검사할 수 있다. 이 기능은 `systemPropertiesMode` 설정을 통해 조절할 수 있다.

### Example: the PropertyOverrideConfigurer

`PropertyOverrideConfigurer`는 또다른 bean factory post-processor로 `PropertyPlaceholderConfigurer`와 비슷하다. 하지만 `PropertyPlaceholderConfigurer`와는 반대로 원본 설정은 bean properties로 기본(default) 값을 가지거나 전혀 값을 가지지 않을 수 있다. 만약 Properties 파일이 특정 bean property를 위한 값을 가지고 있지 않을 경우, 기본 context definition이 사용된다. Properties 파일 설정의 각 줄은 다음과 같은 형식이다.

```
beanName.property=value
```

Spring 2.5부터 지원되는 `context namespace`를 사용하면 다음과 같이 설정할 수 있다.

```
<context:property-override location="classpath:override.properties"/>
```

## FactoryBeans를 사용한 확장(Customizing instantiation logic using FactoryBeans)

`org.springframework.beans.factory.FactoryBean` interface를 구현한 객체는 스스로 factory가 된다. `FactoryBean` interface는 Spring IoC container에 객체화 로직을 삽입할 수 있는 방법이다. 만약 복잡한 객체화 코드를 가지고 있어 장황한 XML 설정보다는 Java로 직접 표현하는 것이 더 좋은 경우, 객체화 코드를 가지고 있는 `FactoryBean`를 생성하여 container에서 삽입할 수 있다.

`FactoryBean` interface는 다음 3가지 메소드를 제공한다.



- `Object getObject()`: 생성한 객체를 return한다. 생성된 객체는 공유될 수도 있다.
- `boolean isSingleton()`: 만약 `FactoryBean`이 singleton을 return한다면 `true`이다. 그렇지 않다면 `false`이다. \* `Class` `getObjectType()`: `getObject()` 메소드에 의해 return되는 객체의 타입을 return한다. 만약 미리 알 수 없는 경우에는 `null`을 return한다. Container에게 `FactoryBean`이 생성한 객체가 아닌 `FactoryBean` 그 자체를 요구하는 경우도 있다. 이런 경우, `BeanFactory`의 `getBean` 메소드를 호출할 때 `bean id` 앞에 `'&'`를 붙이면 된다.

## 참고자료

-  Spring Framework - Reference Document / 3.7. Container extension points

개요

설명

ApplicationContext는 BeanFactory를 확장한 것으로 BeanFactory의 기능 외에 아래와 같은 기능을 제공한다.

- The ApplicationContext
  - 개요
  - 설명
    - BeanFactory or ApplicationContext?
    - MessageSources를 사용한 국제화(Internationalization using MessageSources)
    - Event
    - 웹 어플리케이션을 위한 편리한 ApplicationContext 객체화(Convenient ApplicationContext instantiation for web applications)
  - 참고자료

- MessageSource : i18n-sytle로 메시지를 access할 수 있도록 지원한다.
- Access to resources : URL, File 등과 같은 자원을 쉽게 access할 수 있도록 지원한다.
- Event propagation : ApplicationListener interface를 구현한 bean에게 Event를 전달한다.
- Loading of multiple (hierarchical) contexts : 계층 구조의 context를 지원함으로써, 어플리케이션의 웹 레이어 등과 같은 특정 레이어에만 집중적인 context를 작성할 수 있다.

BeanFactory or ApplicationContext

아주 특별한 이유가 없는 한 ApplicationContext를 사용하는 것이 좋다. 다음은 BeanFactory와 ApplicationContext의 기능 비교표이다.

Feature	BeanFactory	ApplicationContext
Bean 객체화/읽음	Yes	Yes
BeanPostProcessor 자동 등록	No	Yes
BeanFactoryPostProcessor 자동 등록	No	Yes
편리한 MessageSource 접근(for i18n)	No	Yes
ApplicationEvent 발송	No	Yes

MessageSources를 사용한 국제화(Internationalization using MessageSources)

본 장의 내용은 Resource를 참조한다.

Event

ApplicationContext는 Event 처리를 위해 ApplicationEvent, ApplicationListener interface를 제공한다. ApplicationListener interface를 구현한 bean은 ApplicationContext에 발생한 모든 ApplicationEvent를 전달받는다. Spring이 제공하는 표준 event는 다음과 같다.

Event	설명
ContextRefreshedEvent	ApplicationContext가 초기화된거나 refresh될 때 발생한다(refresh하기 위해서 ConfigurableApplicationContext interface의 refresh() 메소드를 사용한다). "초기화"라는 단어는, 모든 bean이 load되었고, post-processor bean이 탐지되어 활성화되었으며, singleton 객체가 선택체 화되어, ApplicationContext 객체가 사용가능한 상태에 있다는 것을 의미한다. refresh는 context가 닫혀지지 않은 한, 여러번 발생할 수 있으며, ApplicationContext가 "hot" refresh를 지원해야한다 (XmlWebApplicationContext는 "hot" refresh를 지원하지만, GenericApplicationContext는 지원하지 않 는다).
ContextStartedEvent	ApplicationContext가 시작될 때 발생한다(시작하기 위해서 ConfigurableApplicationContext interface의 start() 메소드를 사용한다). "시작됨(Started)"이란 단어는, 모든 Lifecycle bean이 명시 적인 시작 신호를 받았음을 의미한다. 이 event는 일반적으로 명시적인 정지(stop) 후에, 재시작하 기 위해서 사용되지만, 자동시작(autostart)로 설정되지 않은 컴포넌트를 시작하기 위해서도 사용된 다
ContextStoppedEvent	ApplicationContext가 정지할 때 발생한다(정지하기 위해서 ConfigurableApplicationContext interface의 stop() 메소드를 사용한다). "정지됨(Stopped)"란 단어는, 모든 Lifecycle bean이 명시 적인 정지 신호를 받았음을 의미한다. 정지된 context는 start() 호출을 통해 재시작될 수 있다.
ContextClosedEvent	ApplicationContext가 닫혔을 때 발생한다(닫기 위해서 ConfigurableApplicationContext interface의 close() 메소드를 사용한다). "닫힘(Closed)"란 단어는, 모든 singleton bean이 파괴되었음을 의미한

	다. 닫힌 context는 생명주기의 끝에 도달한 것으로, refresh 되거나 재시작될 수 없다.
RequestHandledEvent	웹에 특화된 event로서, HTTP request가 처리되었음을 알린다(request가 종료된 후에 발송된다). Spring의 DispatcherServlet를 사용하는 웹 어플리케이션인 경우에만 사용할 수 있다.

새로운 event를 구현하는 것도 어렵지 않다. ApplicationContext interface를 구현한 새로운 event 객체를 ApplicationContext의 publishEvent() 메소드를 통해 발행하면 된다. publishEvent() 메소드는 모든 listener가 event 처리를 마칠때까지 block 상태로 있게 된다. 게다가 transaction context가 가능하다면, listener가 event를 받았을 때, 발행모듈의 transaction context 내에서 event를 처리한다.

아래는 예제이다.

```
<bean id="emailer" class="example.EmailBean">
  <property name="blackList">
    <list>
      <value>black@list.org</value>
      <value>white@list.org</value>
      <value>john@doe.org</value>
    </list>
  </property>
</bean>

<bean id="blackListListener" class="example.BlackListNotifier">
  <property name="notificationAddress" value="spam@list.org"/>
</bean>
```

```
public class EmailBean implements ApplicationContextAware {
    private List blackList;
    private ApplicationContext ctx;

    public void setBlackList(List blackList) {
        this.blackList = blackList;
    }

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx = ctx;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent event = new BlackListEvent(address, text);
            ctx.publishEvent(event);
            return;
        }
        // send email...
    }
}
```

```
public class BlackListNotifier implements ApplicationListener {
    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof BlackListEvent) {
            // notify appropriate person...
        }
    }
}
```

## 웹 어플리케이션을 위한 편리한 ApplicationContext 객체화 (Convenient ApplicationContext instantiation for web applications)

BeanFactory가 프로그램적으로 생성되는 것과 반대로, ApplicationContext 객체는 ContextLoader 등을 사용하여 선언적으로 생성된다. 물론 ApplicationContext 객체 역시 프로그램적으로 생성할 수 있다.

ContextLoader에는 ContextLoaderListener와 ContextLoaderServlet가 있다. 둘 다 기능적으로는 같지만, listener 버전은 Servlet 2.3 컨테이너에서 사용할 수 있다. Servlet 2.4 스펙 이후로, servlet context listener는 웹 어플리케이션을 위한 servlet context가 생성되어 첫번째 요청을 처리할 상태가 된 직후 수행된다(그리고 servlet context가 막 종료되었을 때도 수행된다). 따라서 servlet context listener가 Spring ApplicationContext를 초기화할 최적의 장소이다.

ContextLoaderListener를 사용하여 ApplicationContext를 등록하는 방법은 아래와 같다.

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

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

<!-- or use the ContextLoaderServlet instead of the above listener -->
<servlet>
  <servlet-name>context</servlet-name>
  <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
-->
```

Listener는 'contextConfigLocation' 파라미터를 검사한다. 만약 존재하지 않으면 기본적으로 /WEB-INF/applicationContext.xml를 사용할 것이다. 만약 'contextConfigLocation' 파라미터 값이 존재할 경우, 미리 정해놓은 구분자(comma(','), semicolon(';'), 공백문자(whitespace))를 사용하여 파라미터 문자열을 분리한 후, application context를 찾을 것이다. Ant-style path 패턴이 지원된다 : 예를 들어 /WEB-INF/\*Context.xml은 "WEB-INF" 디렉토리에 존재하는 "Context.xml"로 끝나는 이름을 가진 모든 파일을 의미하고, /WEB-INF/\*\*/\*Context.xml은 "WEB-INF" 디렉토리 및 하위 디렉토리에 존재하는 "Context.xml"로 끝나는 이름을 가진 모든 파일을 의미한다.

## 참고자료

-  Spring Framework - Reference Document / 3.8. The ApplicationContext

## 개요

## 설명

Spring 2.5는 Spring의 종속성 삽입을 위해 annotation을 사용할 수 있다. 본질적으로 `@Autowired` annotation은 자동 엮임과 같은 기능을 제공하지만, 좀 더 세밀한 제어와 넓은 사용성을 제공한다. Spring 2.5는 `@Resource`, `@PostConstruct`, `@PreDestroy` 등의 JSR-250 annotation도 지원한다. 물론 최소 Java 5(Tiger)를 사용하는 경우 사용 가능하다. 이들 annotation을 사용하기 위해서는 Spring container에 특정 `BeanPostProcessors`를 등록해야만 한다. 항상 그렇듯이, 이들 `BeanPostProcessors`가 개별적인 bean 정의로 등록될 수도 있지만, 'context' namespace를 사용하여 등록할 수도 있다.

- Annotation-based configuration
  - 개요
  - 설명
    - `@Required`
    - `@Autowired`
    - Qualifier를 사용한 annotation 기반의 자동 엮임(Fine-tuning annotation-based autowiring with qualifiers)
    - `@Resource`
    - `@PostConstruct`와 `@PreDestroy`
  - 참고자료

```
<?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-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config/>

</beans>
```

(위 `<context:annotation-config/>` tag를 사용하면, `AutowiredAnnotationBeanPostProcessor`, `CommonAnnotationBeanPostProcessor`, `PersistenceAnnotationBeanPostProcessor`, `RequiredAnnotationBeanPostProcessor`를 등록해준다.)

## @Required

`@Required` annotation은 bean property setter 메소드에 적용된다.

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;

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

    // ...
}
```

이 annotation은 단순히 bean property가 반드시 설정되어야만 한다는 것을 나타낸다. 즉, bean 정의에 명시적으로 property 값을 선언하거나 자동 엮임을 통해서 설정되어야만 한다는 것을 의미한다. 만약 annotation이 적용된 bean property에 대한 설정이 이루어지지 않는 경우, container는 exception을 던진다.

## @Autowired

`@Autowired` annotation은 "전통적인" setter 메소드에 적용된다.

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;

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

    // ...
}
```

뿐만 아니라, 임의의 이름과 다수의 argument를 가진 메소드에도 적용될 수 있다.

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

또한, @Autowired annotation은 생성자 및 field에도 적용될 수 있다.

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

또한, array 타입의 field나 메소드에 적용함으로써, ApplicationContext에 존재하는 특정 Type의 모든 bean을 field나 메소드에 적용하는 것도 가능하다.

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

Typed collections에도 같은 방식이 적용된다.

```
public class MovieRecommender {
    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

심지어 typed Map 역시 key 타입이 String인 경우 자동역임이 가능하다. Map은 기대한 타입의 모든 bean을 value로 갖게 되고, key는 해당하는 bean의 이름이 된다.

```
public class MovieRecommender {
    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

기본적으로, 자동역임은 대상이 되는 bean이 없을 경우 실패한다. 기본적으로 annotation이 적용된 메소드, 생성자, field는 필수로 간주한다. 아래와 같이 설정하여 기본 행동 방식을 변경할 수 있다.

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;

    @Autowired(required=false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

@Autowired annotation은 잘 알려진 "분석가능한 종속성(resolvable dependencies)"에도 사용될 수 있다 : BeanFactory interface, ApplicationContext interface, ResourceLoader interface, ApplicationEventPublisher interface, MessageSource interface(그리고 이들을 상속한 ConfigurableApplicationContext 또는 ResourcePatternResolver interface)는 특별한 설정 없이 자동적으로 해결(resolve)된다.

## Qualifier를 사용한 annotation 기반의 자동 역임 (Fine-tuning annotation-based autowiring with qualifiers)

Type을 이용한 자동역임은 대상이 다수가 발생할 수 있기 때문에, 선택 시 추가적인 제어가 필요하다. 한 방법으로 Spring의 @Qualifier annotation을 사용할 수 있다. 특정 argument를 qualifier와 관련시킴으로써, 타입을 찾을 대상을 좁히고, 각 argument에 해당하는 대상 bean을 선택할 수 있다.

```
public class MovieRecommender {
    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
}
```

@Qualifier annotation은 생성자의 argument 및 메소드의 parameter 각각에 적용할 수 있다.

```
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;  
    }  
    // ...  
}
```

일치하는 bean 정의는 아래 예제에서 찾을 수 있다. Qualifier "main" 값을 가진 bean이 같은 값의 @Qualifier annotation이 있는 생성자 argument로 엮인다.

```
<?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-2.5.xsd  
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-  
        2.5.xsd">  
    <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"/>  
</beans>
```

대체 수단으로, bean 이름을 qualifier로 간주된다. 즉, qualifier element 대신 bean id가 "main"이라면 동일하게 동작한다. 어쨌든, 이름으로 bean을 찾는게 더 좋지만, @Autowired는 기본적으로 type을 기반으로 동작하고 qualifier는 선택적이다. 즉, 타입으로 bean 대상을 줄인 후에 qualifier 또는 bean name으로 대상을 좁힌다. Qualifier 값은 의미적으로 유일한 bean id를 나타내지는 않는다. 좋은 qualifier 값은 "main", "EMEA", "persistent" 등과 같이 bean id가 아닌 특정 컴포넌트의 특징을 표현하는 것이다.

Qualifier는 typed collection에도 적용된다.

## @Resource

Spring은 field나 bean property setter 메소드에 적용된 JSR-250 @Resource annotation을 사용하여 종속성 삽입을 지원한다.

@Resource는 'name' attribute를 가지고, Spring은 그 값을 삽입할 bean 이름으로 인식한다.

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

만약 name이 명시적으로 설정되어 있지 않으면, field나 setter 메소드의 이름으로 부터 name 값을 유추해낸다. Field의 경우, field 명과 같다. Setter 메소드의 경우, bean property 이름과 같다. 아래 예제에서는 "movieFinder" bean이 삽입된다.

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

@Autowired와 비슷하게, @Resource도 대안으로 bean type으로 대상을 찾는다. 뿐만 아니라 잘 알려진 "resolvable dependencies"로 해결한다. 둘 다 명시적으로 name을 설정하지 않은 경우 적용된다. 다음 예에서 customerPreferenceDao field를 위해서 "customerPreferenceDao" 이름을 가진 bean을 먼저 찾는다. 그 다음으로 CustomerPreferenceDao Type의 bean을 찾는다. "context" field는 알려진 해결가능한 종속성 type인 ApplicationContext이 기반하여 삽입된다.

```

public class MovieRecommender {
    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}

```

## @PostConstruct와 @PreDestroy

CommonAnnotationBeanPostProcessor는 @Resource annotation 뿐 아니라 JST-250 *lifecycle* annotation 역시 인식한다.

```

public class CachingMovieLister {

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

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

}

```

## 참고자료

-  Spring Framework - Reference Document / 3.11. Annotation-based configuration



## 개요

## 설명

앞선 대부분의 예제들은 Spring container 안에서 `BeanDefinition`을 생성하기 위한 설정 메타데이터를 명기하기 위해서 XML을 사용해왔다. 이전 section [Annotation-based configuration](#)은 source-level annotation을 사용하여 많은 양의 설정 메타데이터를 제공할 수 있음을 보였다. 이들 예제에서도 "base" bean 정의가 XML 파일에 명시적으로 정의되었다. 이번 section은 classpath를 검색하고, *filter*를 통해대상 컴퍼넌트 (*candidate component*)를 검출하는 방법을 소개한다.

- Classpath scanning for managed components
  - 개요
  - 설명
    - @Component and further stereotype annotations
    - Auto-detection Components
    - Naming autodetected components
    - Providing a scope for autodetected components
    - Providing qualifier metadata with annotations
  - 참고자료

## @Component and further stereotype annotations

Spring 2.0부터 Data Access Object(DAO) 등과 같은 repository를 표시하기 위해서 `@Repository` annotation이 소개되었다. Spring 2.5는 추가적으로 `@Component`, `@Service`, `@Controller` annotation을 제공한다. `@Component`는 Spring이 관리하는 컴포넌트를 위한 포괄적인 stereotype을 나타낸다. `@Repository`, `@Service`, `@Controller`는 좀 더 특별한 사용을 위한 `@Component`의 한 일종이다 (각각 persistence, service, presentation layer의 component를 의미한다).

## Auto-detection Components

Spring은 'stereotyped' class를 자동으로 탐지하고 `ApplicationContext`에 일치하는 `BeanDefinition`을 등록하는 기능을 제공한다. 아래 두 class는 자동 탐지의 대상이 된다.

```
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

```
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

실제로 위 두 class를 자동 탐지하고 상응하는 bean을 등록하기 위해서는, 아래 예제 XML의 `<context:component-scan/>` element의 'basePackage'가 위 두 class의 공통 부모 package이어야 한다(또는 comma(',')로 구분된 list 역시 가능하다).

```
<?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-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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

</beans>
```

## Naming autodetected components

검색 과정에서 component가 자동 탐지되었을 때, bean 이름은 scan이 가지고 있는 `BeanNameGenerator` 전략에 따라 생성된다. 기본적으로 name 값을 가지고 있는 Spring 'stereotype' annotation(`@Component`, `@Repository`, `@Service`, `@Controller`)은 상응하는 bean 정의에게 이름을 제공한다. 만약 이들 annotation이 name이 없거나 또 다른 탐지된 component인 경우, 기본 bean name generator는 class 이름의 첫문자를 소문자로 변환한 값을 return할 것이다. 예를 들어, 아래 예제에서 두개의 component가 탐지되는데 각각의 이름은 'myMovieLister'와 'movieFinderImpl'이다.

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

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

## Providing a scope for autodetected components

일반적으로 Spring 관리 component는 'singleton'이다. 하지만 다른 scope이 필요한 경우가 있다. Spring 2.5는 `@Scope` annotation을 제공한다.

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

## Providing qualifier metadata with annotations

이번 section에서는 자동역임의 대상을 찾을 때 상세한 제어를 제공하기 위해 `@Qualifier` annotation을 사용하는 방법을 설명한다.

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

## 참고자료

- Spring Framework - Reference Document / 3.12. Classpath scanning for managed components

개요

AOP 서비스는 관점지향 프로그래밍(Aспект Oriented Programming: AOP) 사상을 구현하고 지원한다. 실행환경 AOP 서비스는 Spring AOP를 사용한다. 본 장에서는 AOP의 개 및 Spring의 AOP 지원을 중심으로 살펴본다.

- AOP 서비스
  - 개요
  - 설명
    - AOP 개요
    - AOP 주요 개념
    - Spring의 AOP 지원
    - 실행환경 AOP 가이드라인
  - 참고자료

설명

AOP 개요

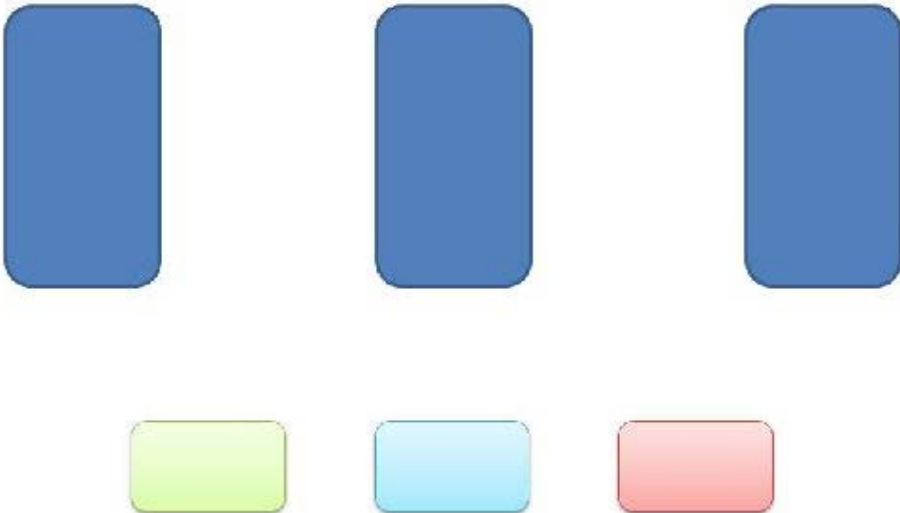
개별 프로그래밍 언어는 프로그램 개발을 위해 고유한 관심사 분리(Separation of Concerns) 패러다임을 갖는다. 예를 들면 절차적 프로그래밍은 상태값을 갖지 않는 연속된 함수들의 실행을 프로그램으로 이해하고 모듈을 주요 분리 단위로 정의한다. 객체지향 프로그래밍은 일련의 함수 실행이 아닌 상호작용하는 객체들의 집합으로 보며 클래스를 주요 단위로 한다. 객체지향 프로그래밍은 많은 장점에도 불구하고, 다수의 객체들에 분산되어 중복적으로 존재하는 공통 관심사가 존재한다. 이들은 프로그램을 복잡하게 만들고, 코드의 변경을 어렵게 한다. 관점 지향 프로그래밍(AOP, Aspect-oriented programming)은 이러한 객체지향 프로그래밍의 문제점을 보완하는 방법으로 핵심 관심사를 분리하여 프로그램 모듈화를 향상시키는 프로그래밍 스타일이다. AOP는 객체를 핵심 관심사와 횡단 관심사 분리하고, 횡단 관심사를 관점(Aspect)이라는 모듈로 정의하고 핵심 관심사와 엮어서 처리할 수 있는 방법을 제공한다.

- 관점(Aspect)은 프로그램의 핵심 관심사에 걸쳐 적용되는 공통 프로그램 영역을 의미한다. 예를 들면 로깅, 인증, 권한확인, 트랜잭션은 비즈니스 기능 구현시에 공통적으로 적용되는 요소이며 하나의 관점으로 정의될 수 있다.
- 핵심 관심사(Core concern)는 프로그램을 작성하려는 핵심 가치와 목적이 드러난 관심 영역으로 보통 핵심 비즈니스 기능에 해당한다.
- 횡단 관심사(Cross-cutting concern)는 핵심 관심에 영향을 주는 프로그램의 영역으로, 로깅과 트랜잭션, 인증처리와 같은 시스템 공통 처리 영역이 해당된다.

다음 그림은 객체지향 프로그래밍 개발에서 핵심 관심사와 횡단 관심사가 하나의 코드로 통합되어 개발된 사례를 보여준다.



객체지향 프로그래밍 코드에 AOP를 적용하면 다음 그림처럼 각 코드에 분산되어 있던 횡단 관심사는 관점으로 분리되어 정의된다. AOP는 엮기(Weaving)라는 방식을 이용하여 분리된 관점을 핵심 관심사와 엮는다.



## AOP 주요 개념

관점 지향 프로그래밍은 횡단 관심사를 분리하고 핵심 관심사와 엮어 사용할 수 있는 방법을 제공하며 다음의 몇 가지 새로운 개념을 포함한다.

### 관점(Aspect)

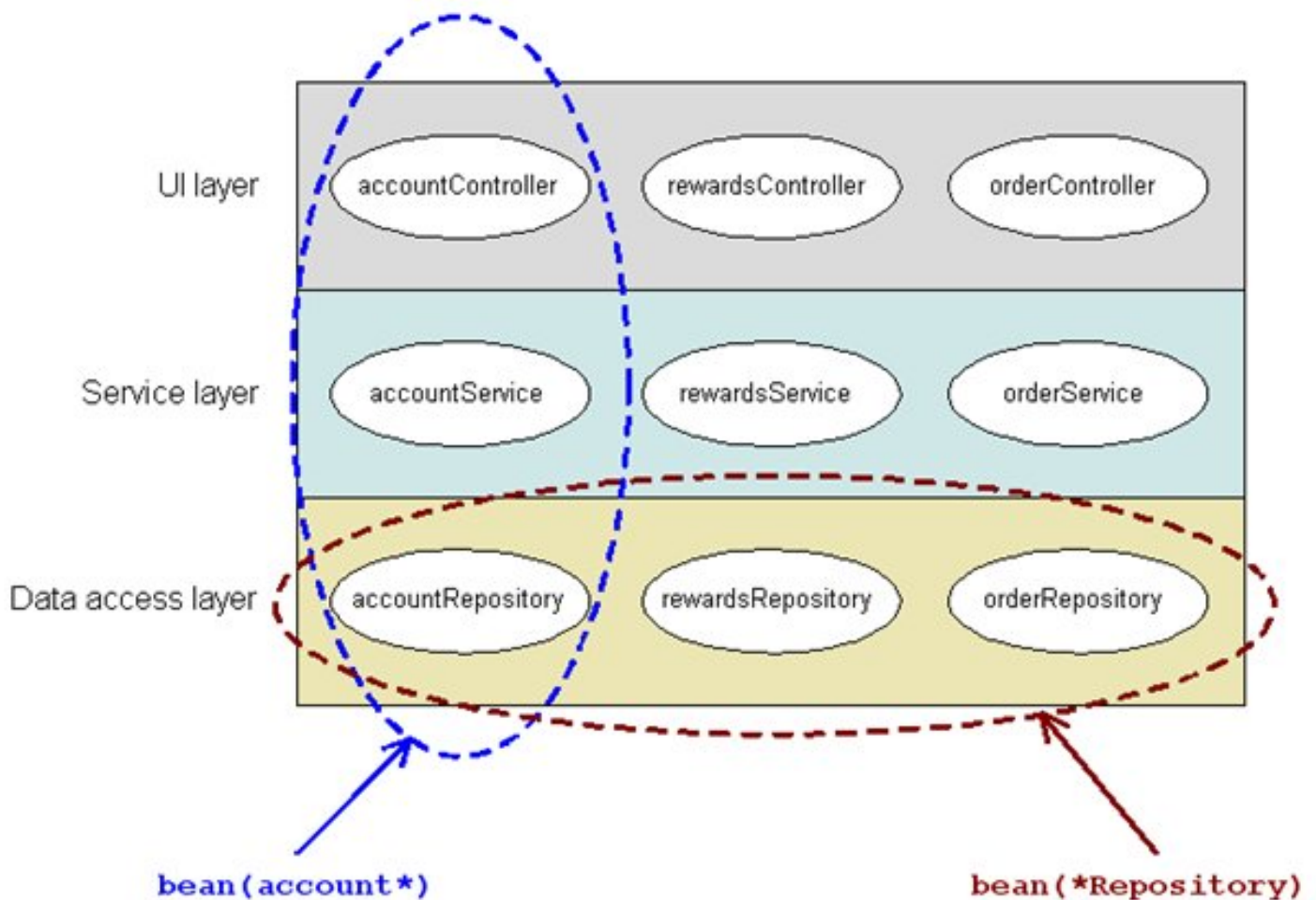
관점은 구현하고자 하는 횡단 관심사의 기능이다.

### 결합점(Join point)

결합점은 관점(Aspect)를 삽입하여 실행 가능한 어플리케이션의 특정 지점을 말한다.

### 포인트컷(Pointcut)

포인트컷은 결합점 집합을 의미한다. 포인트컷은 어떤 결합점을 사용할 것인지를 결정하기 위해 패턴 매칭을 이용하여 룰을 정의한다. 다음 그림은 Spring 2.5에 포함된 `bean()` 포인트컷을 이용하여 종적 및 횡적으로 빈을 선택하는 예제를 보여준다.



## 충고(Advice)

충고(Advice)는 관점(Aspect)의 실제 구현체로 결합점에 삽입되어 동작할 수 있는 코드이다. 충고는 결합점과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다.

- Before advice: joinpoint 전에 수행되는 advice
- After returning advice: joinpoint가 성공적으로 리턴된 후에 동작하는 advice
- After throwing advice: 예외가 발생하여 joinpoint가 빠져나갈때 수행되는 advice
- After (finally) advice: join point를 빠져나가는(정상적이거나 예외적인 반환) 방법에 상관없이 수행되는 advice
- Around advice: joinpoint 전, 후에 수행되는 advice

## 역기(Weaving)

역기는 관점(Aspect)을 대상 객체에 적용하여 새로운 프록시 객체를 생성하는 과정이다. 역기 방식은 다음과 같이 구분된다.

- 컴파일 시 역기: 별도 컴파일러를 통해 핵심 관심사 모듈의 사이 사이에 관점(Aspect) 형태로 만들어진 횡단 관심 코드들이 삽입되어 관점(Aspect)이 적용된 최종 바이너리가 만들어지는 방식이다. (ex. AspectJ, ...)
- 클래스 로딩 시 역기: 별도의 Agent를 이용하여 JVM이 클래스를 로딩할 때 해당 클래스의 바이너리 정보를 변경한다. 즉, Agent가 횡단 관심사 코드가 삽입된 바이너리 코드를 제공함으로써 AOP를 지원하게 된다. (ex. AspectWerkz, ...)
- 런타임 역기: 소스 코드나 바이너리 파일의 변경없이 프록시를 이용하여 AOP를 지원하는 방식이다. 프록시를 통해 핵심 관심사를 구현한 객체에 접근하게 되는데, 프록시는 핵심 관심사 실행 전후에 횡단 관심사를 실행한다. 따라서 프록시 기반의 런타임 역기의 경우 메소드 호출시에만 AOP를 적용할 수 있다는 제한점이 있다. (ex. Spring AOP, ...)

## 도입(Introduction)

도입(Introduction)은 새로운 메소드나 속성을 추가한다. Spring AOP는 충고(Advice)를 받는 대상 객체에 새로운 인터페이스를 추가할 수 있다.

## AOP 프록시(Proxy)

AOP 프록시(Proxy)는 대상 객체(Target Object)에 Advice가 적용된 후 생성되는 객체이다.

## 대상 객체(Target Object)

대상 객체는 충고(Advice)를 받는 객체이다. Spring AOP는 런타임 프록시를 사용하므로 대상 객체는 항상 프록시 객체가 된다.

## Spring의 AOP 지원

스프링은 프록시 기반의 런타임 Weaving 방식을 지원한다. 스프링은 AOP 구현을 위해 다음 세가지 방식을 제공한다. 이 중 @AspectJ 어노테이션과 XML 스키마를 이용한 AOP 방식을 상세히 살펴본다.

- @AspectJ 어노테이션을 이용한 AOP 구현
- XML Schema를 이용한 AOP 구현
- 스프링 API를 이용한 AOP 구현

## 실행환경 AOP 가이드라인

\* 실행환경 AOP 가이드라인

## 참고자료

-  Spring 2.5 Reference Documentation
-  Spring bean() Pointcut

개요

@AspectJ는 Java 5 어노테이션을 사용한 일반 Java 클래스로 관점(Aspect)를 정의하는 방식이다. @AspectJ 방식은 AspectJ 5 버전에서 소개되었으며, Spring은 2.0 버전부터 AspectJ 5 어노테이션을 지원한다. Spring AOP 실행환경은 AspectJ 컴파일러나 직조기(Weaver)에 대한 의존성이 없이 @AspectJ 어노테이션을 지원한다.

- @AspectJ 어노테이션을 이용한 AOP 지원
  - 개요
  - 설명
    - @AspectJ 설정하기
    - 관점(Aspect) 정의하기
    - 포인트컷(Pointcut) 정의하기
    - 충고(Advice) 정의하기
    - 관점(Aspect) 실행하기
  - 참고자료

설명

@AspectJ 설정하기

@AspectJ를 사용하기 위해서 다음 코드를 Spring 설정에 추가한다.

```
<aop:aspectj-autoproxy/>
```

관점(Aspect) 정의하기

클래스에 @Aspect 어노테이션을 추가하여 Aspect를 생성한다. @Aspect 설정이 되어 있는 경우 Spring은 자동적으로 @Aspect 어노테이션을 포함한 클래스를 검색하여 Spring AOP 설정에 반영한다.

```
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectUsingAnnotation {
    ...
}
```

포인트컷(Pointcut) 정의하기

포인트컷은 결합점(Join points)을 지정하여 충고(Advice)가 언제 실행될지를 지정하는데 사용된다. Spring AOP는 Spring 빈에 대한 메소드 실행 결합점만을 지원하므로, Spring에서 포인트컷은 빈의 메소드 실행점을 지정하는 것으로 생각할 수 있다.

다음 예제는 egovframework.rte.fdl.aop.sample 패키지 하위의 Sample 명으로 끝나는 클래스의 모든 메소드 수행과 일치할 'targetMethod' 라는 이름의 pointcut을 정의한다.

```
@Aspect
public class AspectUsingAnnotation {
    ...
    @Pointcut("execution(public * egovframework.rte.fdl.aop.sample.*Sample.*(..))")
    public void targetMethod() {
        // pointcut annotation 값을 참조하기 위한 dummy method
    }
    ...
}
```

포인트컷 지정자(Designators)

Spring에서 포인트컷 표현식에 사용될 수 있는 지정자는 다음과 같다. 포인트컷은 모두 public 메소드를 대상으로 한다.

- execution: 메소드 실행 결합점(join points)과 일치시키는데 사용된다.
- within: 특정 타입에 속하는 결합점을 정의한다.
- this: 빈 참조가 주어진 타입의 인스턴스를 갖는 결합점을 정의한다.
- target: 대상 객체가 주어진 타입을 갖는 결합점을 정의한다.
- args: 인자가 주어진 타입의 인스턴스인 결합점을 정의한다.
- @target: 수행중인 객체의 클래스가 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @args: 전달된 인자의 런타임 타입이 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @within: 주어진 어노테이션을 갖는 타입 내 결합점을 정의한다.
- @annotation: 결합점의 대상 객체가 주어진 어노테이션을 갖는 결합점을 정의한다.

포인트컷 표현식 조합하기

포인트컷 표현식은 '&&', '||' 그리고 '!' 를 사용하여 조합할 수 있다.

```

@Pointcut ("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut ("within(com.xyz.someapp.trading..*)")
private void inTrading() {}

@Pointcut ("anyPublicOperation() && inTrading()")
private void tradingOperation() {}

```

## 포인트컷 정의 예제

Spring AOP에서 자주 사용되는 포인트컷 표현식의 예를 살펴본다.

Pointcut	선택된 Joinpoints
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지과 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 객체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.xyz.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점
bean(accountRepository)	"accountRepository" 빈
!bean(accountRepository)	"accountRepository" 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 "Repository"로 끝나는 모든 빈
bean(accounting/*)	이름이 "accounting/"로 시작하는 모든 빈
bean(*dataSource)    bean(*DataSource)	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

## 충고(Advice) 정의하기

충고(Advice)는 관점(Aspect)의 실제 구현체로 포인트컷 표현식과 일치하는 결합점에 삽입되어 동작할 수 있는 코드이다. 충고는 결합점과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다.

### Before advice

Before advice는 @Before 어노테이션을 사용한다.

다음은 Before 충고를 사용하는 예제이다. Before 충고인 beforeTargetMethod() 메소드는 targetMethod()로 정의된 포인트컷 전에 수행된다.



```

@Aspect
public class AspectUsingAnnotation {
    @Before("targetMethod()")
    public void beforeTargetMethod(JoinPoint thisJoinPoint) {
        Class clazz = thisJoinPoint.getTarget().getClass();
        String className = thisJoinPoint.getTarget().getClass().getSimpleName();
        String methodName = thisJoinPoint.getSignature().getName();
        System.out.println("AspectUsingAnnotation.beforeTargetMethod executed.");
        System.out.println(className + "." + methodName + " executed.");
    }
}

```

## After returning advice

After returning 충고는 정상적으로 메소드가 실행될 때 수행된다. After returning 충고는 @AfterReturning 어노테이션을 사용한다.

다음은 After returning 충고를 사용하는 예제이다. afterReturningTargetMethod() 충고는 targetMethod()로 정의된 포인트컷 후에 수행된다. targetMethod() 포인트컷의 실행 결과는 retVal 변수에 저장되어 전달된다.

```

@Aspect
public class AspectUsingAnnotation {
    @AfterReturning(pointcut = "targetMethod()", returning = "retVal")
    public void afterReturningTargetMethod(JoinPoint thisJoinPoint,
        Object retVal) {
        System.out.println("AspectUsingAnnotation.afterReturningTargetMethod executed." +
            " return value is [" + retVal + "]);
    }
}

```

## After throwing advice

After throwing 충고는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다. After throwing 충고는 @AfterThrowing 어노테이션을 사용한다.

다음은 After throwing 충고를 사용하는 예제이다. afterThrowingTargetMethod() 충고는 targetMethod()로 정의된 포인트컷에서 예외가 발생한 후에 수행된다. targetMethod() 포인트컷에서 발생한 예외는 exception 변수에 저장되어 전달된다. 예제에서는 전달 받은 예외를 한번 더 감싸서 사용자가 쉽게 알아 볼 수 있도록 메시지를 설정하여 반환한다.

```

@Aspect
public class AspectUsingAnnotation {
    @AfterThrowing(pointcut = "targetMethod()", throwing = "exception")
    public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,
        Exception exception) throws Exception {
        System.out.println("AspectUsingAnnotation.afterThrowingTargetMethod executed.");
        System.out.println("에러가 발생했습니다.", exception);

        throw new BizException("에러가 발생했습니다.", exception);
    }
}

```

## After (finally) advice

After (finally) 충고는 메소드 수행 후 무조건 수행된다. After (finally) 충고는 @After 어노테이션을 사용한다. After 충고는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다. 리소스 해제와 같은 작업이 해당된다.

다음은 After (finally) 충고를 사용하는 예제이다. afterTargetMethod() 충고는 targetMethod()로 정의된 포인트컷 이후에 수행된다.

```

@Aspect
public class AspectUsingAnnotation {
    @After("targetMethod()")
    public void afterTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AspectUsingAnnotation.afterTargetMethod executed.");
    }
}

```

## Around advice

Around 충고는 메소드 수행 전후에 수행된다. Around 충고는 @Around 어노테이션을 사용한다.

다음은 Around 충고를 사용하는 예제이다. aroundTargetMethod() 충고는 파라미터로 ProceedingJoinPoint을 전달하며 proceed() 메소드 호출을 통해 대상 포인트컷을 실행한다. 포인트컷 수행 결과값인 retVal을 Around 충고 내에서 변환하여 반환할 수 있음을 보여준다.

```

@Aspect
public class AspectUsingAnnotation {
    @Around("targetMethod()")
    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)
        throws Throwable {
        System.out.println("AspectUsingAnnotation.aroundTargetMethod start.");
    }
}

```



```

        long time1 = System.currentTimeMillis();
        Object retVal = thisJoinPoint.proceed();

        System.out.println("ProceedingJoinPoint executed. return value is [" + retVal + "]");

        retVal = retVal + "(modified)";
        System.out.println("return value modified to [" + retVal + "]");

        long time2 = System.currentTimeMillis();
        System.out.println("AspectUsingAnnotation.aroundTargetMethod end. Time(" + (time2 - time1) + ")");
        return retVal;
    }
}

```

## 관점(Aspect) 실행하기

앞서 정의한 관점(Aspect)가 정상적으로 동작하는지 확인하기 위해 테스트 코드를 이용해 확인해 본다. AnnotationAspectTest 클래스는 대상 메소드 수행시 예외없이 정상 실행하는 경우와 예외 발생의 경우를 구분해서 테스트 한다.

### 정상 실행의 경우

testAnnotationAspect() 함수는 대상 메소드가 정상 수행되는 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AnnotationAdviceSample 클래스의 someMethod() 메소드는 before, after returning, after finally, around 충고(Advice)가 적용된다.

```

public class AnnotationAspectTest {
    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspect() throws Exception {
        SampleVO vo = new SampleVO();
        ..
        String resultStr = annotationAdviceSample.someMethod(vo);
        assertEquals("someMethod executed.(modified)", resultStr);
    }
}

```

테스트 코드를 수행한 결과 로그는 다음과 같다.

```

AspectUsingAnnotation.beforeTargetMethod executed.
AspectUsingAnnotation.aroundTargetMethod start.
ProceedingJoinPoint executed. return value is [someMethod executed.]
return value modified to [someMethod executed.(modified)]
AspectUsingAnnotation.aroundTargetMethod end. Time(78)
AspectUsingAnnotation.afterTargetMethod executed.
AspectUsingAnnotation.afterReturningTargetMethod executed. return value is [someMethod executed.(modified)]

```

콘솔 로그 출력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드
- @Around (대상 메소드 수행 후)
- @After(finally)
- @AfterReturning

주의할 점은 @Around 충고는 대상 메소드의 반환 값(return value)을 변경 가능하지만, After returning 충고는 반환 값을 참조 가능하지만 변경할 수 없다.

### 예외 발생의 경우

testAnnotationAspectWithException() 함수는 대상 메소드에 오류가 발생한 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AnnotationAdviceSample 클래스의 someMethod() 메소드는 before, after throwing, after finally, around 충고(Advice)가 적용된다.

```

public class AnnotationAspectTest {
    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspectWithException() throws Exception {
        SampleVO vo = new SampleVO();
        // exception 을 발생시키도록 플래그 설정
        vo.setForceException(true);
        ..
        try {
            // vo 의 forceException 플래그가 true 이면 - / by zero 상황을 강제로 처리함
            resultStr = annotationAdviceSample.someMethod(vo);
            fail("exception 을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");
        } catch (Exception e) {
            ..
        }
    }
}

```

```
}  
}  
}
```

테스트 코드를 수행한 결과 로그는 다음과 같다.

```
AspectUsingAnnotation.beforeTargetMethod executed.  
AspectUsingAnnotation.aroundTargetMethod start.  
AspectUsingAnnotation.afterTargetMethod executed.  
AspectUsingAnnotation.afterThrowingTargetMethod executed.  
에러가 발생했습니다.  
java.lang.ArithmeticException: / by zero  
...
```

콘솔 로그 출력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드 (ArithmeticException 예외가 발생한다)
- @After(finally)
- @AfterThrowing

예외가 발생하더라도 after 로 정의한 충고(Advice)는 수행되는 것을 확인할 수 있다. After Throwing 충고(Advice)는 에러 메시지를 재설정하고 새로운 예외를 생성하여 전달할 수 있다.

## 참고자료

-  Spring 2.5 Reference Documentation

## 개요

Java 5 버전을 사용할 수 없거나, XML 기반 설정을 선호한다면, Spring 2.0 이상에서 제공하는 XML 스키마 기반의 AOP를 사용할 수 있다. Spring은 관점(Aspect) 정의를 지원하기 위해 "aop" 네임스페이스를 제공한다. @AspectJ를 이용한 AOP 지원에서 사용된 포인트컷 표현식과 충고(Advice) 유형은 XML 스키마 기반 AOP 지원에도 동일하게 제공된다.

- XML 스키마 기반 AOP 지원
  - 개요
  - 설명
    - 관점(Aspect) 정의하기
    - 포인트컷(Pointcut) 정의하기
    - 충고(Advice) 정의하기
    - 관점(Aspect) 실행하기
  - 참고자료

## 설명

## 관점(Aspect) 정의하기

Spring 어플리케이션 컨텍스트에서 빈으로 정의된 일반 Java 개체는 관점(Aspect)으로 정의될 수 있다. 관점(Aspect)은 <aop:aspect> 요소를 사용하여 정의한다.

```
<bean id="adviceUsingXML" class="egovframework.rte.fdl.aop.sample.AdviceUsingXML" />
<aop:config>
  <aop:aspect ref="adviceUsingXML">
    ...
  </aop:aspect>
</aop:config>
```

관점(Aspect)로 정의된 aBean은 Spring 빈처럼 설정되고 의존성 주입이 될 수 있다.

## 포인트컷(Pointcut) 정의하기

포인트컷은 결합점(Join points)을 지정하여 충고(Advice)가 언제 실행될지를 지정하는데 사용된다. Spring AOP는 Spring 빈에 대한 메소드 실행 결합점만을 지원하므로, Spring에서 포인트컷은 빈의 메소드 실행점을 지정하는 것으로 생각할 수 있다.

다음 예제는 egovframework.rte.fdl.aop.sample 패키지 하위의 Sample 명으로 끝나는 클래스의 모든 메소드 수행과 일치할 'targetMethod' 라는 이름의 pointcut을 정의한다. 포인트컷은 <aop:config> 요소 내에 정의한다. 포인트컷 표현식은 AspectJ 포인트컷 표현 언어와 동일하게 사용할 수 있다.

```
<aop:config>
  <aop:pointcut id="targetMethod"
    expression="execution(* egovframework.rte.fdl.aop.sample.*Sample.*(..))" />
</aop:config>
```

## 충고(Advice) 정의하기

충고(Advice)는 관점(Aspect)의 실제 구현체로 포인트컷 표현식과 일치하는 결합점에 삽입되어 동작할 수 있는 코드이다. 충고는 결합점과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다. @AspectJ를 이용한 AOP와 동일하게 5종류의 충고(Advice)를 지원한다.

## Before advice

Before advice는 <aop:aspect> 요소 내에서 <aop:before> 요소를 사용하여 정의한다.

다음은 before 충고를 정의하는 XML의 예제이다. before 충고인 beforeTargetMethod() 메소드는 targetMethod()로 정의된 포인트컷 전에 수행된다.

```
<aop:aspect ref="adviceUsingXML">
  <aop:before pointcut-ref="targetMethod" method="beforeTargetMethod" />
</aop:aspect>
```

다음은 before 충고를 구현하고 있는 클래스이다. before 충고를 수행하는 beforeTargetXML()메소드는 해당 포인트컷을 가진 클래스명과 메소드명을 출력한다.

```
public class AdviceUsingXML {
    ...
    public void beforeTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AdviceUsingXML.beforeTargetMethod executed.");

        Class clazz = thisJoinPoint.getTarget().getClass();
        String className = thisJoinPoint.getTarget().getClass().getSimpleName();
        String methodName = thisJoinPoint.getSignature().getName();

        System.out.println(className + "." + methodName + " executed.");
    }
}
```

```

    }
    ...
}

```

## After returning advice

After returning 충고는 정상적으로 메소드가 실행될 때 수행된다. After 충고는 `<aop:aspect>` 요소 내에서 `<aop:after-returning>` 요소를 사용하여 정의한다.

다음은 After returning 충고를 사용하는 예제이다. `afterReturningTargetMethod()` 충고는 `targetMethod()`로 정의된 포인트컷 후에 수행된다. `targetMethod()` 포인트컷의 실행 결과는 `retVal` 변수에 저장되어 전달된다.

```

<aop:aspect ref="adviceUsingXML">
  <aop:after-returning pointcut-ref="targetMethod"
                    method="afterReturningTargetMethod" returning="retVal" />
</aop:aspect>

```

다음은 After returning 충고를 구현하고 있는 클래스이다. After returning 충고를 수행하는 `afterReturningTargetMethod()` 메소드는 해당 포인트컷의 반환값을 출력한다.

```

public class AdviceUsingXML {
    ...
    public void afterReturningTargetMethod(JoinPoint thisJoinPoint,
        Object retVal) {
        System.out.println("AdviceUsingXML.afterReturningTargetMethod executed." +
            "return value is [" + retVal + "]");
    }
    ...
}

```

## After throwing advice

After throwing 충고는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다. After 충고는 `<aop:aspect>` 요소 내에서 `<aop:after-throwing>` 요소를 사용하여 정의한다.

다음은 After throwing 충고를 사용하는 예제이다. `afterThrowingTargetMethod()` 충고는 `targetMethod()`로 정의된 포인트컷에서 예외가 발생한 후에 수행된다. `targetMethod()` 포인트컷에서 발생한 예외는 `exception` 변수에 저장되어 전달된다.

```

<aop:aspect ref="adviceUsingXML">
  <aop:after-throwing pointcut-ref="targetMethod"
                    method="afterThrowingTargetMethod" throwing="exception" />
</aop:aspect>

```

다음은 After throwing 충고를 구현하고 있는 클래스이다. After throwing 충고를 수행하는 `afterReturningTargetMethod()` 메소드는 전달 받은 예외를 한번 더 감싸서 사용자가 쉽게 알아 볼 수 있도록 메시지를 설정하여 반환한다.

```

public class AdviceUsingXML {
    ...
    public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,
        Exception exception) throws Exception {
        System.out.println("AdviceUsingXML.afterThrowingTargetMethod executed.");
        System.err.println("에러가 발생했습니다.", exception);
        throw new BizException("에러가 발생했습니다.", exception);
    }
    ...
}

```

## After (finally) advice

After (finally) 충고는 메소드 수행 후 무조건 수행된다. After 충고는 `<aop:aspect>` 요소 내에서 `<aop:after>` 요소를 사용하여 정의한다.

다음은 After (finally) 충고를 사용하는 예제이다. `afterTargetMethod()` 충고는 `targetMethod()`로 정의된 포인트컷의 정상종료 및 예외 발생의 경우 모두에 대해 수행된다. 보통은 리소스 해제와 같은 작업을 수행한다.

```

<aop:aspect ref="adviceUsingXML">
  <aop:after pointcut-ref="targetMethod" method="afterTargetMethod" />
</aop:aspect>

```

다음은 After (finally) 충고를 구현하고 있는 클래스이다. After (finally) 충고를 수행하는 `afterTargetMethod()` 메소드는 after 충고가 수행됨을 표시하는 메시지를 출력한다.

```

public class AdviceUsingXML {
    ...
    public void afterTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AdviceUsingXML.afterTargetMethod executed.");
    }
    ...
}

```

## Around advice

Around 충고는 메소드 수행 전후에 수행된다. After 충고는 <aop:aspect> 요소 내에서 <aop:around> 요소를 사용하여 정의한다. Around 충고는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다. 리소스 해제와 같은 작업이 해당된다.

```
<aop:aspect ref="adviceUsingXML">
  <aop:around pointcut-ref="targetMethod" method="aroundTargetMethod" />
</aop:aspect>
```

다음은 Around 충고를 구현하고 있는 클래스이다. aroundTargetMethod() 충고는 파라미터로 ProceedingJoinPoint를 전달하며 proceed() 메소드 호출을 통해 대상 포인트컷을 실행한다. 포인트컷 수행 결과값인 retVal을 Around 충고 내에서 변환하여 반환할 수 있음을 보여준다.

```
public class AdviceUsingXML {
    ...
    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)
        throws Throwable {
        System.out.println("AdviceUsingXML.aroundTargetMethod start.");
        long time1 = System.currentTimeMillis();
        Object retVal = thisJoinPoint.proceed();

        System.out.println("ProceedingJoinPoint executed. return value is ["
            + retVal + "]);

        retVal = retVal + "(modified)";
        System.out.println("return value modified to [" + retVal + "]);

        long time2 = System.currentTimeMillis();
        System.out.println("AdviceUsingXML.aroundTargetMethod end. Time("
            + (time2 - time1) + ")");
        return retVal;
    }
    ...
}
```

## 관점(Aspect) 실행하기

앞서 정의한 관점(Aspect)가 정상적으로 동작하는지 확인하기 위해 테스트 코드를 이용해 확인해 본다. AdviceTest클래스 대상 메소드 수행시 예외없이 정상 실행하는 경우와 예외 발생의 경우를 구분해서 테스트 한다.

### 정상 실행의 경우

testAdvice() 함수는 대상 메소드가 정상 수행되는 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AdviceSample 클래스의 someMethod() 메소드는 before, after returning, after finally, around 충고(Advice)가 적용된 "

```
public class AdviceTest {
    @Resource(name = "adviceSample")
    AdviceSample adviceSample;

    @Test
    public void testAdvice() throws Exception {
        SampleVO vo = new SampleVO();
        ...
        String resultStr = adviceSample.someMethod(vo);
        assertEquals("someMethod executed.(modified)", resultStr);
    }
}
```

테스트 코드를 수행한 결과 로그는 다음과 같다.

```
AdviceUsingXML.beforeTargetMethod executed.
AdviceSample.someMethod executed.
AdviceUsingXML.aroundTargetMethod start.
AdviceUsingXML.afterReturningTargetMethod executed. return value is [someMethod executed.]
AdviceUsingXML.afterTargetMethod executed.
ProceedingJoinPoint executed. return value is [someMethod executed.]
return value modified to [someMethod executed.(modified)]
AdviceUsingXML.aroundTargetMethod end. Time(12)
```

콘솔 로그 출력력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드
- @AfterReturning
- @After(finally)
- @Around (대상 메소드 수행 후)

주의할 점은 @Around 충고는 대상 메소드의 반환 값(return value)를 변경 가능하지만, After returning 충고는 반환 값 참조 가능하지만 변경할 수 없다.

## 예외 발생의 경우

testAnnotationAspectWithException() 함수는 대상 메소드에 오류가 발생한 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AnnotationAdviceSample 클래스의 someMethod() 메소드는 before, after throwing, after finally, around 충고(Advice)가 적용된다.

```
public class AdviceTest {
    @Resource(name = "adviceSample")
    AdviceSample adviceSample;

    @Test
    public void testAdviceWithException() throws Exception {
        SampleVO vo = new SampleVO();
        // exception 을 발생시키도록 플래그 설정
        vo.setForceException(true);
        ..
        try {
            // vo 의 forceException 플래그가 true 이면 - / by zero 상황을 강제로 처리함
            resultStr = adviceSample.someMethod(vo);

            fail("exception 을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");
        } catch (Exception e) {
            ..
        }
    }
}
```

테스트 코드를 수행한 결과 로그는 다음과 같다.

```
AdviceUsingXML.beforeTargetMethod executed.
AdviceSample.someMethod executed.
AdviceUsingXML.aroundTargetMethod start.
AdviceUsingXML.afterThrowingTargetMethod executed.
에러가 발생했습니다.
java.lang.ArithmeticException: / by zero
...
```

콘솔 로그 출력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드 (ArithmeticException 예외가 발생한다)
- @AfterThrowing
- @After(finally)

예외가 발생하더라도 after 로 정의한 충고(Advice)는 수행되는 것을 확인할 수 있다. After Throwing 충고(Advice)는 에러 메시지를 재설정하고 새로운 예외를 생성하여 전달할 수 있다.

## 참고자료

-  Spring 2.5 Reference Documentation

개요

전자정부 실행환경은 XML Schema에 기반한 AOP 방법을 사용하며, 예외처리와 트랜잭처리에 적용하였다. XML Schema에 기반한 AOP 방법은 @AspectJ Annotation 기반 비해 횡단 관심사에 대한 설정관계를 파악하기 유리하다.

- 실행환경 AOP 가이드라인
  - 개요
  - 설명
    - 예외 처리
    - 트랜잭션 처리
  - 참고자료

설명

예외 처리

실행환경은 DAO에서 발생한 Exception을 받아 Service단에서 처리할 수 있다. 실행환경에서 추가로 제공하는 Exception은 다음과 같다.

- EgovBizException: 업무에서 Checked Exception인 경우에 공통으로 사용하는 Exception이다. 개발자가 특정한 오류에 대해서 throw하여 특정 메시지를 전달하고자 하는 경우에는 processException() 메소드를 이용하도록 한다.
- ExceptionTransfer: AOP 기능을 이용하여 ServiceImpl 클래스에서 Exception이 발생한 경우(after-throwing인 경우)에 trace()메소드에서 처리한다. 내부적으로 EgovBizException인지 RuntimeException(ex.DataAccessException)인지 구분하여 throw한다. ExceptionTransfer는 내부적으로 DefaultExceptionHandlerManager 클래스에 의해서 정의된 패턴에 대해서 Handler에 의해서 동작한다.

관점(Aspect) 정의

예외 처리를 위한 Spring 설정 파일(resources/egovframework.spring/context-aspect.xml) 내에 관점(Aspect) 클래스를 bean으로 정의한 뒤, 해당 관점(Aspect)에 대한 포인트컷과 충고(Advice)를 정의한다.

```
<bean id="exceptionTransfer" class="egovframework.rte.fdl.cmmn.aspect.ExceptionTransfer">
...
</bean>
<aop:config>
  <aop:pointcut id="serviceMethod"
    expression="execution(* egovframework.rte.sample.service..*Impl.*(..))" />
  <aop:aspect ref="exceptionTransfer">
    <aop:after-throwing throwing="exception"
      pointcut-ref="serviceMethod" method="transfer" />
  </aop:aspect>
</aop:config>
...
</beans>
```

ExceptionTransfer는 egovframework.rte.sample.service 패키지 내에 속한 모든 클래스 중 클래스명이 Impl로 끝나는 클래스의 메소드 실행시 발생한 예외를 처리하는 역할을 수행한다.

충고(Advice) 정의

충고(Advice)로 정의된 ExceptionTransfer 클래스는 실행환경 소스코드에 포함되어 있다. ExceptionTransfer 클래스는 예외가 발생된 경우 내부적으로 예외처리 설정 파일에 명시된 ExceptionHandler를 호출하는 기능을 한다. ExceptionTransfer 클래스의 코드 일부는 다음과 같다.

```
public class ExceptionTransfer {
    ...
    public void transfer(JoinPoint thisJoinPoint, Exception exception) throws Exception {
        log.debug("execute ExceptionTransfer.transfer ");

        Class clazz = thisJoinPoint.getTarget().getClass();
        Locale locale = LocaleContextHolder.getLocale();

        // BizException 인 경우는 이미 메시지 처리 되었음. 로그만 기록
        if (exception instanceof EgovBizException) {
            log.debug("Exception case :: EgovBizException ");

            EgovBizException be = (EgovBizException) exception;
            getLog(clazz).error(be.getMessage(), be.getCause());

            // Exception Handler 에 발생된 Package 와 Exception 설정.
            processHandling(clazz, exception, pm, exceptionHandlerServices, false);

            throw be;
        } else if (exception instanceof RuntimeException) {
            log.debug("RuntimeException case :: RuntimeException ");

            RuntimeException be = (RuntimeException) exception;
            getLog(clazz).error(be.getMessage(), be.getCause());

            // Exception Handler 에 발생된 Package 와 Exception 설정.
        }
    }
}
```

```

        processHandling(clazz, exception, pm, exceptionHandlerServices, true);

        if (be instanceof DataAccessException) {
            log.debug("RuntimeException case :: DataAccessException ");
            DataAccessException sqlEx = (DataAccessException) be;
            // throw processException(clazz, "fail.data.sql",
            // new String[] {
            // Integer.toString(((SQLException) sqlEx
            // .getCause()).getErrorCode()),
            // ((SQLException) sqlEx.getCause())
            // .getLocalizedMessage() }, sqlEx, locale);
            throw sqlEx;
        }

        throw be;
    } else if (exception instanceof FdlException) {
        log.debug("FdlException case :: FdlException ");

        FdlException fe = (FdlException) exception;
        getLog(clazz).error(fe.getMessage(), fe.getCause());

        throw fe;
    } else {
        log.debug("case :: Exception ");

        getLog(clazz).error(exception.getMessage(), exception.getCause());

        throw processException(clazz, "fail.common.msg", new String[] {}, exception, locale);
    }
}
}
}

```

## 트랜잭션 처리

실행환경에서 트랜잭션 설정은 "resources/egovframework.spring/context-transaction.xml" 파일을 참조한다. 다음은 context-transaction.xml 설정 파일을 일부이다.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <!-- 트랜잭션 관리자를 설정한다. -->
    <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 트랜잭션 Advice를 설정한다. -->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="*" rollback-for="Exception"/>
        </tx:attributes>
    </tx:advice>

    <!-- 트랜잭션 Pointcut를 설정한다. --->
    <aop:config>
        <aop:pointcut id="requiredTx"
            expression="execution(* egovframework.rte.sample..impl.*Impl.*(..))"/>
        <aop:advisor advice-ref="txAdvice"
            pointcut-ref="requiredTx" />
    </aop:config>

</beans>

```

- txAdvice는 메소드에서 예외 발생시 트랜잭션 롤백을 수행한다.
- requiredTx는 egovframework.rte.sample 패키지 하위 impl 패키지에서 Impl로 끝나는 모든 클래스의 메소드를 포인트컷으로 지정한다.

## 참고자료

- 실행환경 Exception Handling 서비스
- 실행환경 Transaction 서비스



개요

리소스를 활용하여 가장 많이 사용하는 메시지 제공 서비스를 살펴본다. 메시지 제공 서비스는 미리 정의된 파일에서 메시지를 읽어 들인 후, 오류 발생시 또는 안내 메시지를 제공하기 위해 키값에 해당하는 메시지를 가져오는 기능을 제공한다.

- Resource 서비스
  - 개요
  - 설명
    - Message Basic
    - Message Locale
    - Message Parameter
  - 참고자료

설명

Message Basic

메시지를 활용하기 위한 기본 설정 및 활용에 대해서 예제를 중심으로 설명한다.

Configuration

```
<bean name="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="useCodeAsDefaultMessage">
    <value>true</value>
  </property>
  <property name="basenames">
    <list>
      <value>egovframework-message</value>
    </list>
  </property>
</bean>
```

위의 설정에서 "egovframework-message" 로 지정한 파일은 실제로는 egovframework-message.properties 로 정의되어 있다. 파일의 위치를 지정하는 방법이 여러가지가 가능한데 그 설정에 대한 것은 [4.참고자료](#) 참조.

Sample Source

```
//egovframework-message.properties에 정의된 메시지 내용.
resource.basic.msg1=message1

@Resource (name="messageSource")
MessageSource messageSource ;

String getMsg = messageSource.getMessage("resource.basic.msg1" , null , Locale.getDefault() );
assertEquals("Get Message Success!", getMsg , "message1");
```

위의 소스를 보면 messageSource.getMessage를 이용하여 Message를 얻는 것을 확인 할 수 있다.

Message Locale

동일한 메시지 키를 가지고 언어별로 별도로 설정 관리하여 사용자에게 따라서 사용자에게 맞는 언어로 메시지를 제공할 수 있다.

Configuration

```
<bean name="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="useCodeAsDefaultMessage">
    <value>true</value>
  </property>
  <property name="basenames">
    <list>
      <value>egovframework-message-locale</value>
    </list>
  </property>
</bean>
```

위의 설정에서 "egovframework-message-locale" 로 지정한 파일을 egovframework-message-locale\_ko.properties,egovframework-message-locale\_en.properties로 정의하고 동일한 메시지에 해당하는 메시지를 달리 지정한다.

Properties File

```
//egovframework-message-locale_ko.properties 파일 내용
resource.locale.msg1=메시지1

//egovframework-message-locale_en.properties 파일 내용
resource.locale.msg1=en_message1
```

위에서 resource.locale.msg1 라는 키에 다른 메시지를 설정한 것을 확인할 수 있다. 위와 같이 설정하면 locale 정보에 따라서 메시지를 제공받을 수 있다.

## Sample Source

```
//egovframework-message.properties에 정의된 메시지 내용.
resource.basic.msg1=message1

String getMsg = messageSource.getMessage("resource.locale.msg1" , null , Locale.KOREAN );
assertEquals("Get Message Success!", getMsg , "메시지1");

String getMsg = messageSource.getMessage("resource.locale.msg1" , null , Locale.ENGLISH );
assertEquals("Get Message Success!", getMsg , "en_message1");
```

위에서 Locale정보에 따라서 추출되는 메시지의 내용이 다른 것을 확인할 수 있다.

## Message Parameter

프로그램 수행중에 발생하는 메시지를 추가하여 제공 할 수 있는데 그것에 대한 사용 방법은 설정은 위와 동일하고 Properties 파일에 아래와 같이 설정한다.

## Properties File

```
resource.basic.msg3=message {0} {1}
```

위에서 {0},{1}로 정의된 부분에 추가 메시지를 입력하여 제공 받을 수 있다. 위의 설정을 활용하는 샘플은 아래와 같다.

## Sample Source

```
Object[] parameter = { new String("1") , new Integer(2) };

String getMsg = messageSource.getMessage("resource.basic.msg3" , parameter , Locale.getDefault() );
assertEquals("Get Message Success!", getMsg , "message 1 2");
```

위에서 parameter에 1과 2를 지정하여 getMessage의 두번째 인자에 넣고 호출하면 리턴 메시지로 "message 1 2"를 얻는 것을 확인 할 수 있다.

## 참고자료

-  Spring Resource