- Spring Core -

프레임워크란?

- 프레임워크는 다른 소프트웨어 프로젝트가 개발될 수 있는 뼈대 구조
- 지원 프로그램, 라이브러리, 언어, 다른 소프트웨어 구성 요소들을 엮어 주는 소프트웨어 등을 포함
- 플랫폼도 프레임워크의 일종이라고 볼 수 있음
- 프로그램 개발을 위한 부분이며 완전한 소프트웨어 실행 환경이 되지 않음
- 플랫폼은 프로그램 실행환경을 포함하는 개념

스프링 프레임워크

- 오픈 소스 프레임워크
  - ✓ Rod Johnson 창시
    - Expert one-on-one J2EE Design - Development, 2002, Wrox
    - Expert one-on-one J2EE Development without EJB, 2004, Wrox
  - ✓ 엔터프라이즈 어플리케이션 개발의 복잡성을 줄여주기 위한 목적
  - ✓ EJB 사용으로 수행되었던 모든 기능을 일반 POJO(Plain Old Java Object) 를 사용해서 가능하게 함. (light weight container)
  - ✓ http://www.springframework.org
- 주요 개념
  - ✓ 의존성 주입(lightweight dependency injection)
  - ✓ 관점 지향 컨테이너(aspect-oriented container)

스프링 탄생 배경

- EJB의 단점으로 등장하게 된 스프링
- 최초 서블릿이 출현하면서, 웹 기반 애플리케이션에서 자바 확산
- 트랜잭션, 보안 등을 제공하는 EJB가 나타나면서 자바는 엔터프라이즈 애플리케이션을 구축하는 데 필요한 기본 기술로 자리 매김.

① 세션 빈이나 엔티티 빈의 코드를 변경해서 테스트해야 할 경우, 컴파일하고 알맞은 포맷으로 묶은 뒤 EJB컨테이너에 배포를 해야만 변경한 코드에 대한 테스트가 가능 → 개발속도를 향상시키는데 한계

② 반드시 EJB스펙에 정의된 인터페이스에 따라 코드를 작성하도록 제약하고 있다. → 개발자가 기존에 작성한 POJO를 변경해야 하는 단점.

③ 엔티티빈이 수행하는 데이터베이스 맵핑이나 메시지 드리븐 빈이 제공하는 메시지 처리 등을 사실 EJB가 아닌 다른 기술들을 통해서 구현 → 컨테이너 없이 테스트가 어렵다!, 개발 속도가 저하된다!

- Rod Johnson의 “Export One-on-One J2EE Development without EJB”
- 이 책을 통해 EJB를 사용하지 않고 엔터프라이즈 어플리케이션을 개발하는 방법을 소개하였고, 이것이 스프링 프레임워크의 모태가 되었음

Java Bean, POJO(Plain Old Java Object)

- 자바 빈(Java Bean)이란 무엇인가?
  - ✓ 개발도구에서 가시적으로 조작이 가능하고 또한 재사용이 가능한 소프트웨어 컴포넌트 (http://ko.wikipedia.org/wiki/자바빈즈)
  - ✓ 디폴트 생성자: 자바빈 파라미터가 없는 디폴트 생성자를 갖고 있어야 한다. 툴이나 프레임워크에서 리플렉션을 이용해 오브젝트를 생성하기 때문에 필요
  - ✓ 프로퍼티: 자바빈이 노출하는 이름을 가진 속성을 프로퍼티라고 한다. 프로퍼티는 set으로 시작하는 수정자 메소드(setter)와 get으로 시작하는 접근자 메소드(getter)를 이요해 수정 또는 조회 가능

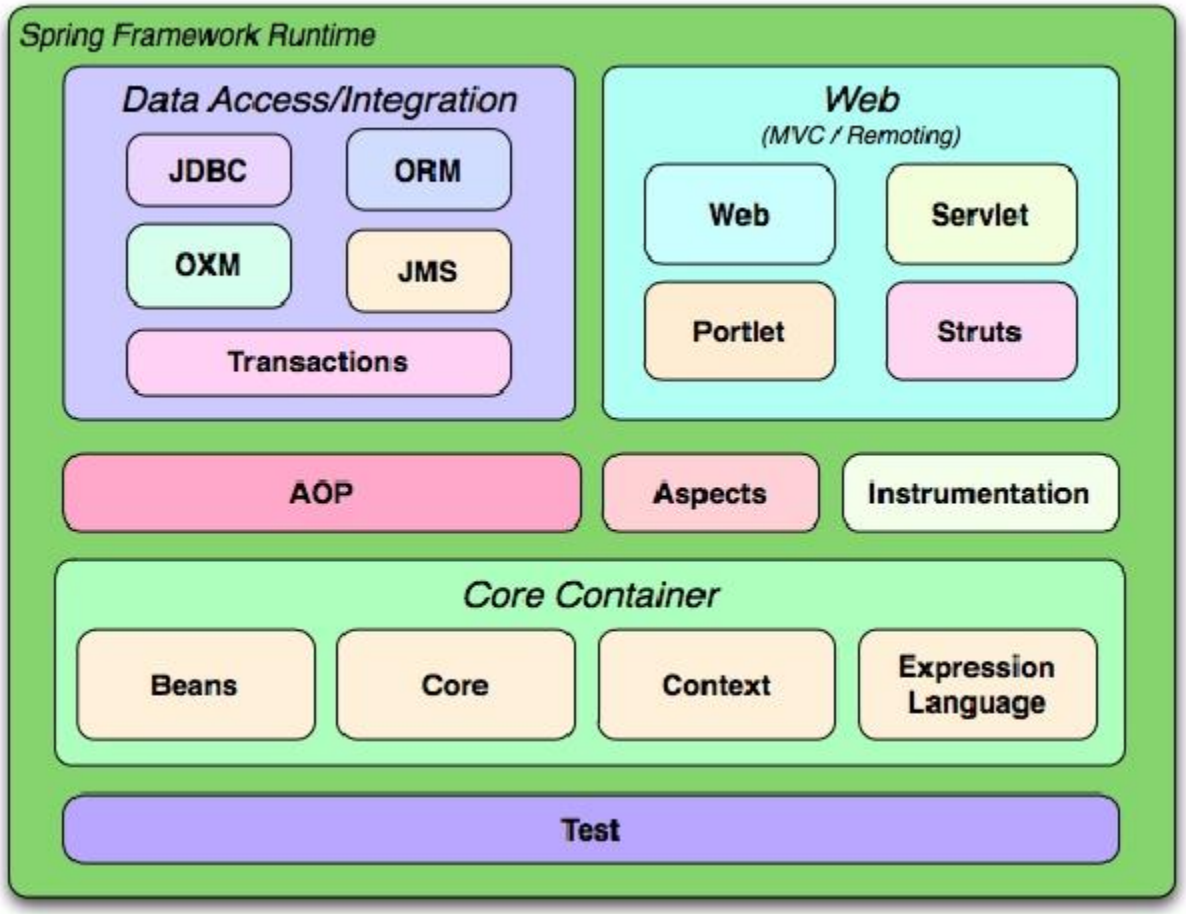
- POJO(Plain Old Java Object)란 무엇인가?
  - ✓ Martin Fowler(http://www.martinfowler.com/bliki/POJO.html)
  - ✓ Servlet과 EJB와 같이 특정 Interface(Contracts)에 종속적이지 않은 모든 자바 클래스
  - ✓ 일반적으로 우리들이 흔히 이야기하는 자바 빈은 모두 POJO라고 이야기할 수 있음

Spring 장점

- 경량 컨테이너 – 객체의 라이프 사이클 관리, JEE 구현을 위한 다양한 API제공
- DI (Dependency Injection) 지원
- AOP (Aspect Oriented Programming) 지원
- POJO (Plain Old Java Object) 지원
- JDBC를 위한 다양한 API 지원
- Transaction 처리를 위한 일관된 방법제공
- 다양한 API와의 연동 지원

스프링 모듈

- 스프링은 20여개의 하위 모듈로 구성
- 핵심: 코어 컨테이너, 데이터 액세스/통합, 웹, AOP, 테스트 프레임워크 등)



출처: Spring Documentation: http://docs.spring.io/spring/docs/3.1.0.M1/spring-framework-reference/html/overview.html#overview-dependency-injection

모듈: 코어 컨테이너

- 구성 요소: Core, Beans, Context, Expression Language
- Core, Beans
  - ✓ 스프링 프레임워크의 가장 핵심 영역
  - ✓ 팩토리 패턴을 사용하여 구현한 BeanFactory 클래스가 핵심 –
    싱글톤(Singleton:객체가 하나만 존재) 개발자가 고민할 필요가 없음
- Context
  - ✓ ApplicationContext 인터페이스를 통해 Core, Beans에 접근 기능을 제공
  - ✓ JNDI 레지스트리와 비슷한 형태로 객체들에 접근할 수 있는 경로를 제공
  - ✓ 리소스 로딩, 이벤트 전파, EJB, JMX 등의 접근도 가능하도록 구성
- Expression Language
  - ✓ 런타임시에 객체 그래프 조작 및 쿼리에 사용되는 표현 언어
  - ✓ 이름에 의해 프로퍼티 값 세팅, 메소드 호출, 컨텍스트 접근, 객체 검색 등을 제공
  - ✓ JSP 2.1 명세서의 통합 EL(Expression Language)와 통합 가능

JNDI: Java Naming and Directory Interface

모듈: 데이터 액세스/통합

- 구성 요소: JDBC, ORM, OXM, Transaction
- JDBC
  - ✓ 기존의 지루한 데이터베이스 코딩을 없앨 수 있도록 JDBC 추상화 레이어 제공
- ORM(Object Relation Mapping, 객체 관계 맵핑)
  - ✓ JPA, JDO, Hibernate, MyBatis와 같은 객체-관계 맵핑과 통합할 수 있는 기능 제공
  - ✓ 트랜잭션 관리 등과 같이 스프링에 제공하는 모든 기능과 통합
- OXM(Object XML Mapping)
  - ✓ JAXB, Castor, XML Beans, XSTream 구현체를 통해 객체(Object)와 XML을 변환
  - ✓ RESTful 서비스를 만들거나 XML 통신을 수행해야 할 경우 유용하게 사용될 수 있음
- Transaction
  - ✓ 모든 POJO에 대해 프로그래밍적과 선언적인 트랜잭션 관리를 지원

JNDI: Java Naming and Directory Interface

모듈: 웹

- 구성 요소: Web, Web-Servlet, Web-Struts, Web-Portlet 모듈
- Web
  - ✓ 서블릿 리스너, 서블릿 컨테스트를 사용하여 IoC 컨테이너를 초기화
  - ✓ 멀티파트 파일 업로드와 같은 기능들을 사전에 구성하여 제공
- Web-Servlet
  - ✓ 웹 애플리케이션에 대한 Spring MVC 구현체를 포함하는 모듈
  - ✓ 도메인 모델과 웹 폼 사이의 완전한 분리 기능을 제공하고 통합
- Web-Struts
  - ✓ 아파치 스트럿츠 2.0과 스프링을 통합할 수 있는 모듈.
  - ✓ Spring 3.0 부터는 Depreciated(더 이상 지원안함) 상태
- Web-Portlet
  - ✓ Web-Servlet 모듈 기능을 복제하여 포틀릿 환경을 만드는 데 사용되는 MVC 구현체

JNDI: Java Naming and Directory Interface

모듈: AOP, Instrumentation, Test

- AOP(Aspect Oriented Programming)
  - ✓ 메소드 인터셉터, 포인트컷 등을 사용하여 기능적으로 분리되어야 하는 구현체를 제공
  - ✓ Aspects 모듈은 AspectJ 통합 기능을 제공
- Instrumentation
  - ✓ 애플리케이션 서버 내에서 사용되는 클래스 로더 구현체에 대한 바이트 코드 조작
- Test
  - ✓ Junit 혹은 TestNG를 활용하여 스프링 컴포넌트를 테스트
  - ✓ Application Context와 컨텍스트 캐싱을 활용
  - ✓ Mock Objects(모의(가짜) 객체)를 사용하여 테스트 영역을 분리 가능

JNDI: Java Naming and Directory Interface

프레임워크 규칙

- 다운로드
  - ✓ http://spring.io/downloads/community
  - ✓ 3.0이후의 Naming Convension: **`org.springframework.*-<version>.jar`**
- Maven 사용
  - ✓ artifactId: spring-*-<version>.jar
  - ✓ groupId: org.springframework

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.0.RELEASE</version>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

스프링 라이브러리 리포지토리

- S3 Maven Repository

```
<repositories>
   <repository>
      <id>com.springsource.repository.maven.release</id>
      <url>http://maven.springframework.org/release/</url>
      <snapshots><enabled>false</enabled></snapshots>
   </repository>
</repositories>
```

- Milestone Maven Repository

```
<repositories>
   <repository>
      <id>com.springsource.repository.maven.milestone</id>
      <url>http://maven.springframework.org/milestone/</url>
      <snapshots><enabled>false</enabled></snapshots>
   </repository>
</repositories>
```

- Spring Logging -

※ 본 단원의 주제:

Logging은 시스템을 운영하는데 있어 아주 중요한 모듈 중의 하나이다. 스프링에 팀에서 사용하는 로거 선택 방법에 대해 논의한다.

로깅

- 로깅은 스프링에서 아주 중요한 영역
  - ✓ 로깅은 스프링 자체가 아닌 외부 의존성 라이브러리를 사용해야만 함
  - ✓ 개발자 혹은 팀에서 사용하는 로깅 모듈이 대부분 틀림
- 스프링은 명시적으로 commons-logging을 사용
  - ✓ Common-logging을 사용하는 경우 별도의 설정 절차는 필요없음

Commons Logging 사용 안하기

- 아래와 같이 Spring에서 commons-logging을 런타임시에 제거

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.0.RELEASE</version>
      <scope>runtime</scope>
      <exclusions>
         <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
</dependencies>
```

SLF4J 적용 방법

- 팀에서 선호하는 라이브러리가 SLF4J 일 경우 아래의 설정 사용

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.0.RELEASE</version>
      <scope>runtime</scope>
      <exclusions>
         <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>jcl-over-slf4j</artifactId>
      <version>1.5.8</version>
      <scope>runtime</scope>
   </dependency>
   <dependency>
```

Log4J 적용 방법

- 팀에서 선호하는 라이브러리가 Log4j일 경우 아래의 설정 사용

```
<dependencies>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.0.RELEASE</version>
      <scope>runtime</scope>
   </dependency>
   <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
      <scope>runtime</scope>
   </dependency>
</dependencies>
```

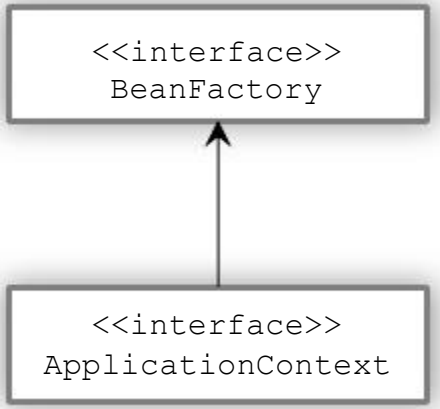
- IoC Container, XML 기반 설정 -

※본 단원의 주제:

Spring의 핵심 기능인 BeanFactory, ApplicationContext 대해 본 단원에서 다룬다. .

스프링 IoC의 핵심

- 스프링 프레임워크가 가진 IoC 컨테이너 기본 패키지
  - ✓ org.springframework.beans
  - ✓ org.springframework.context
  - ✓ 다양한 종류의 인터페이스 클래스들이 존재

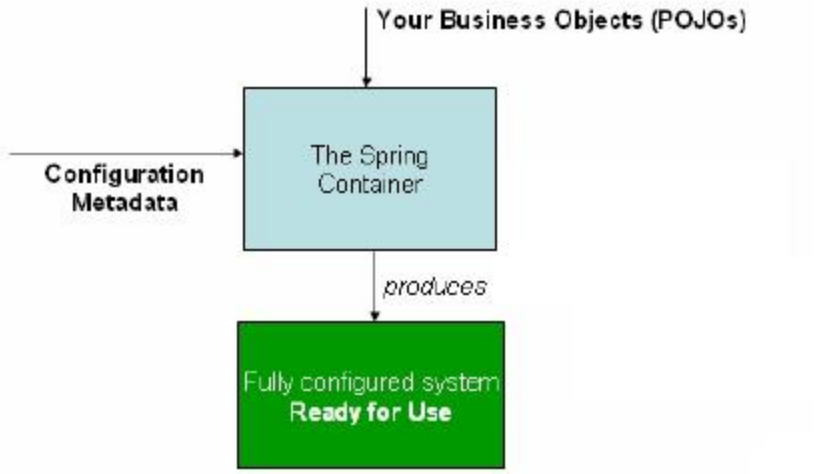


- BeanFactory
  - ✓ 빈 객체를 관리하고 각 빈 객체간의 의존 관계를 설정해 주는 기능을 제공
- ApplicationContext
  - ✓ BeanFactory + AOP, 메시지, i18n 등의 다양한 추가 기능
  - ✓ ClassPathXmlApplicationContext, FileSystemXmlApplicationContext
  - ✓ WebApplicationContext: Application당 하나씩 생성하며, 주요 구현 클래스는 XmlWebApplicationContext

http://docs.spring.io/spring/docs/3.1.4.RELEASE/javadoc-api/

org.springframework.context.ApplicationContext

- 역할
  - ✓ 빈의 객체 생성, 설정, 조립 등을 수행하는 핵심 영역
  - ✓ XML, 자바 어노테이션, 자바 코드를 통해 메타데이터 설정 가능
  - ✓ 객체 간의 상호 의존성과 애플리케이션에 대한 인젝션 기능을 수행
- 일반적인 구현
  - ✓ ClassPathXmlApplicationContext 또는 FileSystemXmlApplicationContext를 통해 생성
  - ✓ 빈 설정이 많지 않을 경우에는 어노테이션 등을 사용하나, 설정이 복잡해질 경우에는 XM
    을 이용하여 설정



메타데이터 설정 방법

- 메타데이터 설정 방식
  - ✓ XML 기반 설정: 최초부터 사용되던 유형으로 일반적으로 이 방법을 사용
  - ✓ 어노테이션 기반 설정: Spring 2.5부터 사용가능
  - ✓ 자바 기반 설정: 프로그래밍을 통해 설정하며 Spring 3.0부터 사용가능
- XML 기반 설정
  - ✓ 본 과정에서 사용될 기본적인 설정 방식
  - ✓ 최상위<beans/> 엘리먼트를 활용하여 빈의 정보를 구성

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
  <!-- more bean definitions go here -->
</beans>
```

컨테이너 초기화

- ApplicationContext 생성자를 활용하여 설정 파일의 위치를 알려줌으로서 생성
  - ✓ CLASSPATH, 로컬 파일 시스템과 같은 외부 리소스로부터 설정 로드 가능
  - ✓ 아래는 클래스 패스에 services.xml, daos.xml이 있어야 함

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

- Resource 클래스를 통한 생성
  - ✓ 다양한 종류의 자원을 동일한 방식으로 통일하여 설정을 로드 가능
  - ✓ UrlResource, ClassPathResource, FileSystemResource, ServletContextResource, InputStreamResource, ByteArrayResource

```
Resource resource = new ClassPathResource("applicationContext.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

객체 인젝션 - 생성자

- 객체 또는 값을 생성자를 통해 주입
- <constructor-arg> : <bean>의 하위태그로 설정한 bean 객체 또는 값을 생성자를 통해 주입하도록 설정
  - ✓ 설정 방법 : <ref>,<value>와 같은 하위태그를 이용하여 설정, 속성을 이용해 설정

```
package x.y;

public class Foo {

  // 생성자를 통해 주입
  public Foo(Bar bar, Baz baz) {
     // ...
  }
}
```

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

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

</beans>
```

객체 인젝션 – 생성자 예제

- ExampleBean에 int와 String 값을 주입

```
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;
  }
}
```

```
<!-- Type을 이용한 방법 -->
<bean id="exampleBean"class="examples.ExampleBean">
<constructor-arg type="int" value="7500000"/>
<constructor-arg type="java.lang.String" value="42"/>
</bean>
<!– Index를 이용한 방법 -->
<bean id="exampleBean"class="examples.ExampleBean">
<constructor-arg index="0" value="7500000"/>
<constructor-arg index="1" value="42"/>
</bean>
<!-- Name를 이용한 방법 -->
<bean id="exampleBean"class="examples.ExampleBean">
<constructor-arg name="years" value="7500000"/>
<constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

객체 인젝션 – 프로퍼티(Property) 이용

- 생성자가 아닌 프로퍼티(Setter)를 사용하며 Setter Injection이란 용어를 사용
  - ✓ private String id 선언이 있을 경우 public void setId(String id)
  - ✓네이밍 규칙 : set + 변수명(첫 글자는 대문자)

```
<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>
```

- 위의 설정을 유추하였을 때 `org.apache.commons.dbcp.BasicDataSource` 클래스 내에 아래의 메소드가 있어야 함
  - ✓ `setDriverClassName, setUrl, setUsername, setPassword`

객체 인젝션 – 프로퍼티(Property) 이용

- Primitive 타입이 아닌 다른 자바 객체를 주입하고자 하는 경우
  - ✓ property에 애트리뷰트를 활용하여 연결함

```
public class AccountService{
   private AccountDao accountDao = null;
   public setAccountDao (AccountDao dao){…}
}
```

```
<bean id="dao" class="x.y.dao.AccountDao"/>
<bean id="service" class="x.y.service.AccountService">
   <property name="accountDao">
      <ref bean ="dao"/>
   </property >
</bean>
<!-- 또는 -->
<bean id="service" class="x.y.service.AccountService">
   < property name="accountDao" ref="dao">
</bean>
```

객체 인젝션 – Collection

- <property> 또는 <constructor-arg>의 하위 태그로 Collection 값을 설정하는 태그를 이용해 값 주입 설정

태그	컬렉션	설명
<list>	java.util.List	List계열컬렉션값목록전달
<set>	java.util.Set	Set계열컬렉션값목록전달
<map>	java.util.Map	Map계열컬렉션에keyvalue의값목록전달
<props>	java.util.Properties	Properties에key(String)-value(String)의값목록전달

JNDI: Java Naming and Directory Interface

객체 인젝션 – Collection List

- <property> 또는 <constructor-arg>의 하위 태그로 Collection 값을 설정하는 태그를 이용해 값 주입 설정

```
public void setMyList(List list){…}
```

```
<bean id="otherbean" class="x.y.OtherBean"/>
<bean id="myBean" class="x.y.MyVO">
   <property name="myList">
      <list>
         <value>10</value> ->String으로 저장
         <value type="java.lang.Integer">20</value> ->Integer로 저장
         <ref bean="otherbean"/>
      </list>
</property>
</bean>
```

객체 인젝션 – Collection Map

- <key>, <key-ref>와 <value>, <value-ref>로 Map에 값을 주입

```
public void setMyMap(Map map){…}
```

```
<bean id="otherbean" class="vo.OtherBean"/>
<bean id="myBean" class="vo.MyVO">
    <property name="myMap">
        <map>
            <entry key="id" value="abc"/>
            <entry key="other" value-ref="otherbean"/>
        </map>
    </property>
</bean>
```

객체 인젝션 – Collection Properties

- <prop>를 이용해 key-value를 properties에 등록

```
public void setJdbcProperty (Properties props){…}
```

```
<bean id="myDAO" class="vo.DAO">
   <property name="jdbcProperty">
      <props>
         <prop key="driver">JDBC Driver</prop>
         <prop key="url">jdbc:url://127.0.0.1/mydb</prop>
         <prop key="user">dbUser</prop>
         <prop key="pwd">dbPassword</prop>
      </props>
   </property>
</bean>
```

객체 인젝션 – Collection Set

- Set 에 값을 주입

```
public void setMySet(Set props){…}
```

```
<bean id="otherbean" class="vo.OtherBean"/>
<bean id="myBean" class="vo.Bean">
    <property name="mySet">
        <set>
            <value>10</value>
            <value>20<value>
            <ref bean="otherbean"/>
        </set>
    </property>
</bean>
```

빈 객체 생성시 살아 있는 범위

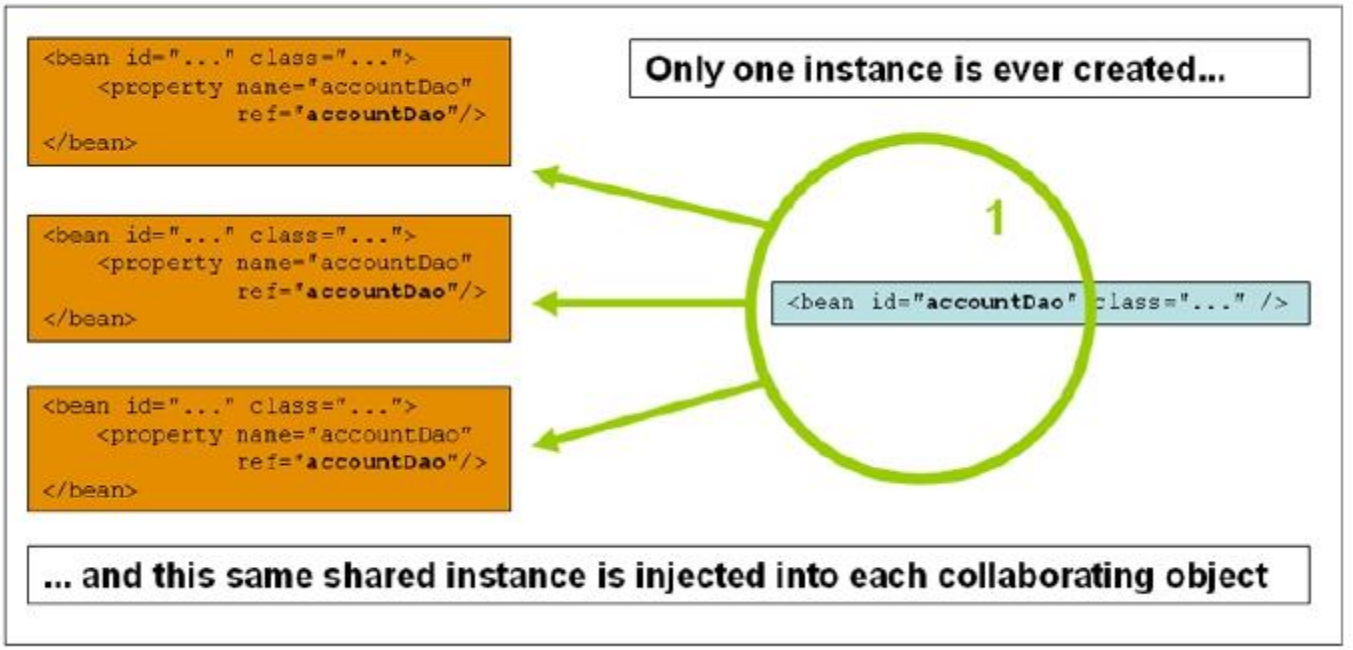
- BeanFactory를 통해 Bean을 요청시 객체생성의 범위(단위)를 설정
- <bean> 의 scope 속성을 이용해 설정

Scope	설명
singleton	컨테이너는하나의빈객체만생성-default
prototype	빈을요청할때마다생성
request	Http요청마다빈객체생성
session	HTTPSession라이프사이클동안하나의빈을사용
global-session	글로벌HTTPSession으로오로지포틀릿컨텍스트를사용할경우에만유효

- request, session은 WebApplicationContext에서만 적용 가능

범위 : singleton

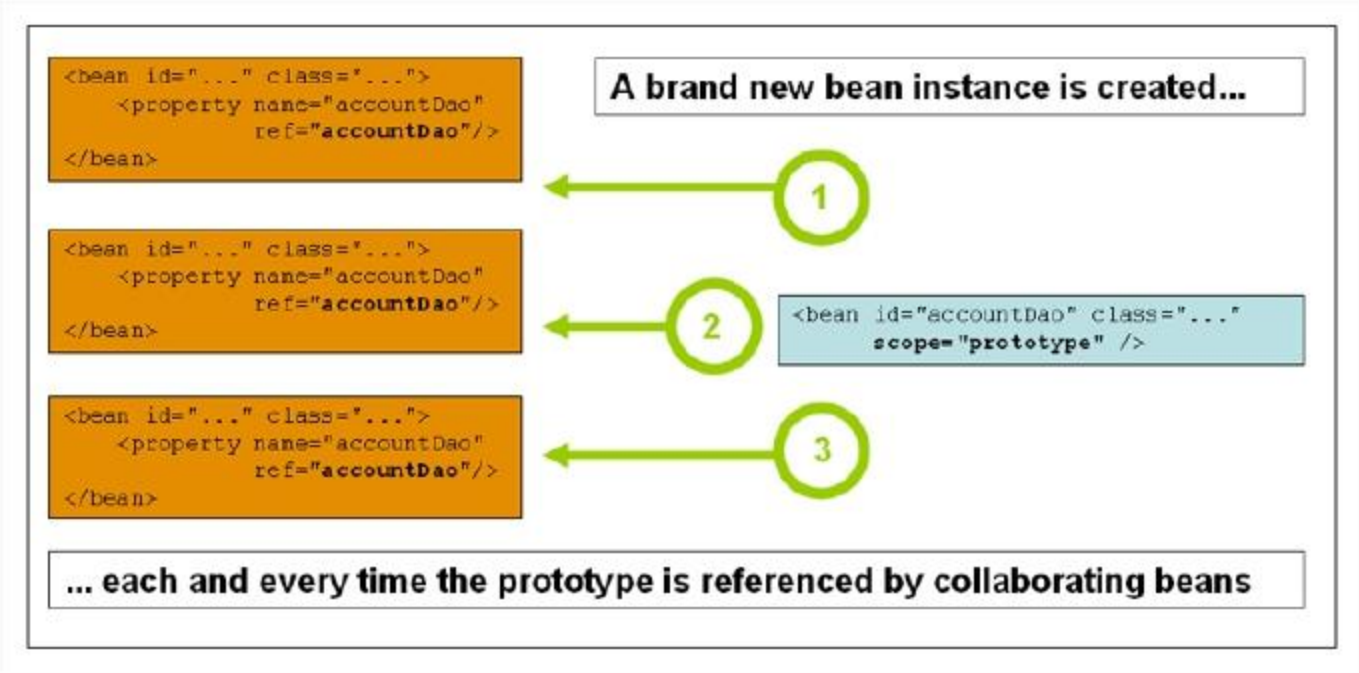
- 컨테이너에 오로지 하나의 공유 싱글톤 인스턴스만 생성
- 싱글톤 인스턴스는 컨테이너 캐시에 저장되며, 캐시된 객체를 반환




```
<bean id="accountService" class="com.foo.DefaultAccountService"/>
<!– singleton이 기본값이므로 지정하지 않으면 싱글톤 빈이 만들어짐-->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

범위 : prototype

- 요청이 들어올 때마다 빈이 생성되도록 설정할 경우 prototype 범위를 사용해야 함
- 컨테이너를 통해 `ctx.getBean()` 메소드 호출시마다 새롭게 생성하여 반환함




```
<!-- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

범위 : request, session, global session

- 오로지 웹 애플리케이션에서만 사용 가능한 범위
- ClassPathXmlApplicationContext에서 사용할 경우 IllegalStateException 발생

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

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

빈 라이프사이클

- 콜백(callback)을 통해 빈이 생성되거나 소멸될 때 작업 수행 가능

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

```
public class ExampleBean {

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

- IoC Container, Annotation 기반 설정 -

※본 단원의 주제:
Spring의 IoC의 인젝션 기능을 어노테이션 기반으로 설명한다.

어노테이션 기반 설정

- XML 설정은 오타 등으로 인한 사용자 오류의 가능성이 높음
- 어노테이션을 사용하여 이클립스 등의 컴파일 타임에 체크 가능
- 클래스 패스를 스캐닝하며 컴포넌트를 찾고 해당 어노테이션 적용된 클래스를 객체화시킴
- Spring 2.5
  - ✓ 객체 생성: @Component, @Repository(since Spring 2.0), @Service, @Controller
  - ✓ 객체 주입: @AutoWired 어노테이션을 사용
- Spring 3.0
  - ✓ 객체 생성: @Configuration, @Bean
  - ✓ 객체 주입: JSR-250(Common Annotations for the Java)를 활용한 @Resource, @PostConstruct, @PreDestroy
  - ✓ 객체 주입: JSR-330(Dependency Injection for Java)를 활용한 @Inject, @Qualifier, @Named, @Provider

클래스 자동 탐지 및 빈 등록

- 설정에 의해 자동으로 어노테이션을 감지하고 ApplicationContext를 통해 BeanDefinitions에 등록 가능
- 자동 스캔을 하기 위해서는 아래의 설정을 사전에 해줘야 함.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

어노테이션 기반 설정 적용

- 어노테이션 기반을 사용하더라도 XML 기반 스프링 설정 파일에 아래의 태그를 등록해야만 어노테이션 인식 가능

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:annotation-config/>

</beans>
```

어노테이션 기반 설정 적용을 main 메소드에서 하기

- XML 파일의 compont-scan, annotation-config 엘리먼트에 해당하는 작업을 main 메소드에서 아래와 같이 코딩으로 적용할 수 있음
- 코딩으로 해결할 경우 XML 파일을 사용하지 않아도 됨

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);\
    myService.doStuff();
}
```

컴포넌트 스캔에 의한 객체 생성

- @Repository, @Service 어노테이션을 클래스 레벨에 선언하여, 객체 생성이 되도록

```
@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
}
```

빈(Bean) 객체에 이름 명명하기

- 스캐닝 작업이 되면 BeanNameGenerator 클래스에 의해 생성된 빈 이름이 적용
- 스캐닝에 의한 빈 생성시 name 을 통한 빈의 이름 정의가 가능 – 대상: @Component, @Repository, @Service, @Controller
- 이름이 지정되지 않으면, 생성기는 클래스 이름을 빈의 이름으로 등록

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

빈(Bean)에 범위(scope) 지정하기

- 앞선 scope(singleton, prototype, request, session, global session)을 어노테이션을 통해 지정 가능
- Spring 2.5 이후 적용 가능

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

객체 인젝션 : @Required

- Setter 프로퍼티 메소드 단위에 설정하며, 해당 빈에서 사용하는 객체가 반드시 다른 객체에 의해 주입되어야 하는 경우 아래와 같이 설정

```
public class SimpleMovieLister {

  private MovieFinder movieFinder;

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

  // ...
}
```

객체 인젝션: @Autowired, @Inject

- Spring 2.5에 @Autowired 속성은 Spring 3.0의 표준 인젝션인 @Inject로 사용 가능
- 해당 어노테이션이 있을 경우 컨테이너는 컨텍스트에서 해당 빈을 찾아 인젝션 시킴
- 생성자, 필드, 메소드에 적용 가능

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

  private CustomerPreferenceDao customerPreferenceDao;

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

객체 인젝션: @Resource

- JSR-250(Common Annotation for Java) 표준으로 필드(변수) 혹은 Setter 메소드에 적용 가능
- name 애트리뷰트 속성을 가지고 있으며, 인젝션시 해당 빈의 이름을 찾아 적용(nam 속성은 없어도 됨)

```
public class SimpleMovieLister {

  @Resource private CustomerPreferenceDao customerPreferenceDao;
  private MovieFinder movieFinder;

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

라이프사이클 : @PostConstruct, @PreDestory

- XML 기반 설정의 init-method, destroy-method에 해당하는 작업을 어노테이션을 통
  수행 가능
- Spring 2.5 이후 부터 지원하고 있으며, 필요에 따라 애플리케이션 넣어서 사용 가능

```
public class CachingMovieLister {

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

- JSR-330을 통한 객체 인젝션 -

※본 단원의 주제:

JSR-330(Dependency Injection for Java)을 통한 객체 인젝션 방법을 설명한다.

JSR-330을 활용하기 위한 pom.xml

- JSR-330을 활용하기 위해서는 구현체가 클래스 패스에 등록되어 있어야 함
- Maven을 이용할 경우 아래의 의존성 삽입을 통한 적용 가능
- @Autowired 대신에 javax.inject.Inject를 사용하여 표준화 시킬 수 있음

```
<dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>
        <version>1</version>
</dependency>
```

객체 주입 : @Inject

- 클래스, 필드, 메소드, 생성자에서 @Inject를 사용 가능
- 특정 이름으로 지정된 빈을 인젝션하고자 하는 경우 @Named 어노테이션을 사용

```
import javax.inject.Inject;
import javax.inject.Named;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

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

@Named 어노테이션은 @Component와 같다

- 스프링의 어노테이션 @Component 대신에 표준인 javax.inject.Named 사용 가능
- 특정 이름으로 지정된 빈을 인젝션하고자 하는 경우 @Named 어노테이션을 사용

```
import javax.inject.Inject;
import javax.inject.Named;

@Named(“movieListener”)
public class SimpleMovieLister {
    private MovieFinder movieFinder;

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

- 프로퍼티 로딩하기 -

※ 본 단원의 주제:
외부 프로퍼티 파일을 통해 필요한 값을 로딩하는 방법을 알아본다.

PropertyPlaceholderConfigurer

- 자바 Properties 포맷을 사용하여 파일 시스템에 존재하는 설정을 빈으로 읽어들이기 위해 PropertyPlaceholderConfigurer 를 사용(JDBC 접속 정보 등)

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

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

```
<!– in Spring 2.5 or laster -->
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

PropertyPlaceholderConfigurer

- 프로그램 내에서 아래와 같은 형태로 직접 인젝션하여 사용 가능

```
@Service(“sapService")
public class SapService {
        protected final Logger logger = LoggerFactory.getLogger(SapService.class);

        @Value("#{contextProperties[‘sap.url']}")
        private String sapUrl;

        @Value("#{contextProperties[‘sap.user.name']}")
        private String username;

        @Value("#{contextProperties[‘sap.user.password']}")
        private String password;

        // do work
}
```