Spring in Action

1장 스프링 속으로

* 스프링 개발 장점 : 자바 개발의 간소화, 테스트의 용이함, 낮은 결합도
  + 간소화 (POJO의 힘) : 개별 Bean의 존재목적과 상이한 코드작성의 최소화
  + DI를 통한 낮은 결합도
    - 기존 : 애플리케이션내 상호 의존하는 두 개 이상의 클래스의 강력한 결합  
      -> 테스트와 재활용이 어렵다 / 이해하기 어렵다 / 코드수정의 여파가 크다
    - 스프링 : 필요한 객체를 인터페이스 형태로 요청하는 객체에 넣어줌(주입)  
      (본문엔 인터페이스 객체를 생성자 주입 constructor injection한 형태로 예를 들음)  
      -> 인터페이스 형태이기 때문에 주입 객체의 변경이 용이하다.  
      -> 인터페이스 형태이기 때문에 모조 구현체를 이용한 테스트가 용이하다

\* Wiring : DI관계의 컴포넌트간(Bean) 연결 (XML / Annotation으로 구성)  
\* Application context : Wiring을 정의하는 스프링 구현체, 다양한 방식이 있음.

* + Aspect oriented programming : 앱 전체에 사용되는 기능을 재사용 가능한 컴포넌트에 담아 사용
    - 각 컴포넌트 본연의 목적을 위한 코드를 수정하지 않고, 추가 기능(특히 횡단관심사)을 가능케 한다.

\* Cross-cutting concerns : 로깅/트랜잭션 관리, 보안 등 애플리케이션 전반에 관여하는 서비스

* + 템플릿을 이용한 상투적인 코드 제거 : 특히 JDBC의 커넥션 열고 닫고 Try / catch등을 간소화
* Spring Container : 스프링 애플리케이션의 객체들의 전체 생명주기를 관리함.
  + 종류
    - Bean Factory : DI에 대한 기본적인 지원을 제공하는 단순한 컨테이너
    - Application Container : 애플리케이션 프레임워크 서비스를 제공 (주로 사용)
    - Application Context :
      * ClassPathXmlApplicationContext : 클래스패스에서
      * FileSystemXmlApplicationContext : 시스템내의 경로에서
      * XmlWebApplicationContext : 웹애플리케이션에 포함된 XML파일에서 컨텍스트 정의 로드
  + 빈의 생명주기 : 생략 / 여러 단계가 있기에 빈 관리의 정교한 커스터마이징이 가능하다.
* 스프링 모듈과 6가지 카테고리
  + 코어 스프링 컨테이너 : Beans / Core / Context / Expression / Context support
    - * + 빈팩토리 / 애플리케이션 컨텍스트 / 이메일 / JNDI액세스 / EJB통합 / 스케줄링 등
  + AOP 모듈 : AOP / Aspects
  + 데이터 액세스와 통합 : JDBC / OXM / Trasaction / ORM / JMS
  + 웹과 리모팅 : Web / Servlet / Portlet / Structs
  + 테스팅 : Test
  + 도구 : Instrument / Instrument Tomcat
* 스프링 포트폴리오 : 책의 범위는 벗어남. 생략.
  + 스프링 웹 플로
  + 스프링 웹 서비스
  + 스프링 시큐리티
  + 스프링 인티그레이션
  + 스프링 배치
  + 스프링 소셜
  + 스프링 모바일
  + 스프링 DM
  + 스프링 LDAP
  + 스프링 리치 클라이언트
  + 스프링 .NET
  + 스프링 – 플렉스
  + 스프링 루
* 스프링 2.5 / 3.0 개선사항 : 생략

2장 빈 와이어링

* XML 빈 설정
  + <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](http://www.springframework.org/schema/beanshttp://www.springframework.org/schema/beans/spring-beans-3.0.xsd)>  
    (여기부터 개별 <bean></bean> 정의 작성)  
    </beans>
  + beans외의 네임스페이스 설정 (총 10개)
    - aop
    - beans
    - context 스프링이 직접 관리하지 않는 객체의 주입 / bean auto-detect, auto-wiring 포함해서 스프링 애플리케이션 컨텍스트 설정하기 위한 엘리먼트 제공
    - jee JNDI와 EJB등의 통합
    - jms 메시지 드리븐 POJO 선언위한 설저
    - lang Groovy, JRuby, BeanShell 스크립트로 구현되는 빈의 선언을 가능하게 함
    - mvc 애너테이션 지향 컨트롤러, 뷰 컨트롤러, 인터셉터 등의 MVC 기능
    - OXM 객체 대 XML 매핑 구조의 설정 지원
    - tx 선언적 트랜젝션 설정 제공
    - util 유틸리티 엘리먼트 선택, 컬렉션을 빈으로 선언하는 기능과 프로퍼티 대치 엘리먼트 지원
  + 빈 선언 방법 (xml내에서 / Juggler Class는 구현했다고 가정)
    - <bean id=”duke” class=”com.springaction.springidol.Juggler”/></bean>
  + 빈 로드 및 실행 (class 내에서)
    - ApplicationContext ctx = new ClassPathXmlApplicationContext(“com/springinaction/springidol/spring-ido.xml”);  
      Performer performer = (Performer) ctx.getBean(“duke”);  
      performer.perform();
  + 빈 선언시 생성자를 통한 주입 방법
    - <bean id=”duke” class=”com.springaction.springidol.Juggler”/>  
      <constructor-arg value=”15”/></bean>
    - <bean id=”duke” class=”com.springaction.springidol.Juggler”/>  
      <constructor-ref value=”sonnet29”/></bean>
    - Value는 주입대상 클래스의 생성자 인자의 변수 / ref는 bean으로 등록한 객체
  + 빈 선언시 팩토리 매서드 또는 정적(static) 메소드 생성
    - <bean id=”theStage” class=”com.springinaction.springidol.Stage” factory-method=”getInstance”/>
    - 생성자 대신에 static getInstance메서드를 인스턴스 생성에 사용한다.(싱글톤 클래스)
  + 빈 선언시 복수 인스턴스 생성
    - <bean id=”ticket” class=”com.springinaction.springidol.ticket” scope=”prototype”/>
    - 그 외의 scope 옵션
      * Singleton(기본값)
      * prototype : 생성시마다 신규 인스턴스
      * request : http 요청당 하나의 인스턴스
      * session 세션당 하나의 인스턴스
      * global-session 전역 http 세션당 하나의 인스턴스 (포틀릿 환경)
  + 빈 선언시 생성 직후 / 소멸 직후 호출 메서드 설정
    - <bean id=”auditorium” class=”com.springinaction.springidol.Auditorium” init-method=”turnOnLights” destroy-method=”turnOffLights”/>
    - 공통적으로 사용시엔 <beans> 엘리먼트 내에 선언하면 된다.
  + 빈 선언시 세터 주입
    - <bean id=”Kenny” class=”com.springinaction.springidol.Instrumentalist”>  
      <property name=”song” value=”Jingle Bells”/>  
      <property name=”instrument” ref=”saxophone”/>  
      </bean>
    - 여기서 주입받는 Instrumentalist의 메서드 파라미터는 instrument 인터페이스 형태. 구현체아님.
    - 내부 빈 주입 : 해당 빈에서만 사용하는 프로퍼티 / 생성자 정의  
      <property name=”instrument”>  
       <bean class=”org.springinaction.springidol.Saxophone”/>  
      </property>  
      생성자 주입도 동일한 방식으로 사용 / 내부에서만 쓰기에 id를 정의할 필요가 없다.
  + p 네임스페이스 사용하기
    - <beans> 내부에 xmlns:p=<http://www.springframework.org/schema/p> 작성
    - 빈 정의시  
      <bean id=”Kenny” class=”com.springinaction.springidol.Instrumentalist”  
      p:song value=”Jingle Bells”  
      p:instrument-ref=”saxophone”/>
  + 컬렉션 빈 선언 java.util.collection 타입
    - 주입받는 쪽에서는 아래와 같은 형태로 파라미터 작성  
      private Collection<Instrument> instrument;  
      public void setInstruments(Collection<Instrument> instruments) {….}
    - <bean id=”hank” class=”com.springinaction.springidol.OneManBand”>  
       <property name=”instruments”>  
       <list>  
       <ref bean=”guitar”/>  
       <ref bean=”cymbal”/>  
       <ref bean=”harmonica”/>  
       </list>  
       </property></bean>  
        
      \* 중복이 없을 경우 list 대신에 <set></set>을 사용해도 됨.
  + map 빈 선언 java.util.Map 타입
    - 주입받는 쪽에서는 아래와 같은 형태로 파라미터 작성  
      private Map<String, Instrument> instruments;  
      public void setInstruments(Map<String, Instrument> instruments) {….}
    - <bean id=”hank” class=”com.springinaction.springidol.OneManBand”>  
       <property name=”instruments”>  
       <map>  
       <entity key=”GUITAR” value-ref=”guitar”/>  
       <entity key=”CYMBAL” value-ref=”cymbal”/>  
       <entity key=”HARMONICA” value-ref=”harmonica”/>  
       </map>  
       </property></bean>  
      \* key-ref / value 도 사용 가능
  + properties 빈 선언 java.util.Properties 타입  
    \* 엔티티와 값이 모두 String이라면 <map> 대신에 <props> 사용
    - 주입받는 쪽에서는 아래와 같은 형태로 파라미터 작성  
      private properties instruments;  
      public void setInstruments(Properties instruments) {….}
    - <property name=”instruments”>  
       <props>  
       <prop key=”GUITAR”>기타기타기타</prop>  
       <prop key=”CYMBAL”>심벌심벌</prop>  
       <prop key=”HARMONICA”>하모하모</prop>  
       </props>  
       </property></bean>
  + null 값을 프로퍼티에 주입
    - <property name=”nullex”><null/></property>
* 스프링 표현식 언어 (Spring Expression Language, SpEL)
  + <property>엘리먼트 value doxmflqbxmdp #{ } 마커를 이용하여 사용한다.
  + 빈의 프로퍼티 참조
    - <bean id=”carl” class=”com.~”>  
       <property name=”song” value=”#{kenny.song}”/>  
      </bean>  
      => carl.setSong(Kenny.getSong());
  + 빈의 메소드 호출
    - <property name=”song” value=”#{songSelector.selectSong()}”/>
    - <property name=”song” value=”#{songSelector.selectSong().toUpperCase}”/>
    - <property name=”song” value=”#{songSelector.selectSong()?.toUpperCase}”/>  
      ?. => 오른쪽 항목 엑세스 이전에 왼쪽 항목이 null이라면 오른쪽을 호출 안함
  + 클래스 메소드나 상수 호출
    - <property name=”multiplier” value=”#{T(java.lang.Math).PI”
    - <property name=”randomNumber” value=”#{T(java.lang.Math).random()}”/>
  + 연산자 사용
    - 일반적인 자바언어의 산수/관계/논리를 지원함
    - 특이한 경우
      * xml 특성상 관계 기호는 아래 표현식을 사용한다.
        + equals == : eq
        + Less than < : lt
        + Less than or equals <= : le
        + Greater than > : gt
        + Greater than or equals >= : ge
      * 삼항연산자 ?: (Elvis operator)
        + <property name=”song” value=”#{Kenny.song !=null ? Kenny.song : ‘Greensleeves’’}”/>
        + <property name=”song” value=”#{Kenny.song ?: ‘Greensleeves’’}”/>  
          -> 위 두개는 같은 결과이나, Kenny.song 중복 호출을 피하는 두번째 표현이 효율적
      * 정규 표현식 : 반환값 true / false
        + <property name=”validEmail” value=”#{admin.email matches ‘[a-zA-Z0-9]’}”/>
  + 컬렉션 엑세스
    - bean 컬렉션 사용 <util:list>
      * <util:list id=”cities”>  
         <bean class=”com.City” p:name=”Chicago” p:state=”IL”/>  
         <bean class=”com.City” p:name=”Atlanta” p:state=”GA”/>  
         <bean class=”com.City” p:name=”Dallas” p:state=”TX”/>  
        </util:list>
      * <property name=”chosenCity” value=”#{cities[2]}”/>
      * <property name=”chosenCity” value=”#{cities.size()}”/>
      * <property name=”chosenCity” value=”#{cities[‘Dallas’]}”/>
      * <property name=”chosenCity” value=”#{cities.?[population gt 10000]}”/>  
        .?[] 조건 만족하는 목록 .^[] 첫번째 값, .#[] 마지막 값
      * <property name=”chosenCity” value=”#{cities.![name]}”/>  
        .![] 프로퍼티에 해당하는 값 목록 반환
      * <property name=”chosenCity” value=”#{cities.![name+’, ‘+state]}”/>  
        프로퍼티 값 결합한 목록으로 반환
      * <property name=”chosenCity” value=”#{cities.?[population gt 10000].![name]}”/>  
        .![] 조건에 일치하고, 프로퍼티에 해당하는 값 목록 반환
    - <util:properties>엘리먼트로 클래스패스의 properties 파일 접근도 가능
    - systemEnvironment
      * <property name=”homePath” value=”#{systemEnvironment[‘HOME’]}”/>

3장 XML 설정 최소화

* 이름으로 Auto Wiring
  + <bean id=”Kenny” class=”com.springinaction.springidol.Instrumentalist” autowire=”byName”>  
    <property name=”song” value=”Jingle Bells”/>  
    <property name=”instrument” ref=”saxophone”/> -> 프로퍼티 이름과 같은 instrument를 bean id로 지정하면 생략이 가능하다.   
    </bean>
* 타입으로 Auto Wiring
  + <bean id=”Kenny” class=”com.springinaction.springidol.Instrumentalist” autowire=”byType”>  
    <property name=”song” value=”Jingle Bells”/>  
    <property name=”instrument” ref=”saxophone”/> -> 프로퍼티 타입과 같은 Instrument 타입을 가진 bean이 있으면 생략이 가능하다.   
    </bean>
  + 할당 가능한 타입의 빈이 둘 이상이라면 예외가 발생한다. 따라서 할당하지 않을 빈의 primary 애트리뷰트 (기본값 true)를 false로 지정하여 기본 오토와이어링 후보가 아님을 설정해야 한다.
    - <bean id=”saxophone” class=”com.~” primary=”false”/>
  + 기본 오토와이어링 후보의 식별이 아닌 아예 후보에서 제외할때는 다음과 같이 설정
    - <bean id=”saxophone” class=”com.~” autowire-candidate=”false”/>
* 생성자 기반 Auto Wiring
  + <bean id=”saxophone” class=”com.~” autowire=”constructor”/>
  + byType과 마찬가지로 생성자가 일치하는 빈은 하나여야만 한다.
* 최적의 Auto Wiring
  + <bean id=”saxophone” class=”com.~” autowire=”autodetect”/>
  + constructor 시도 -> byType 시도
* 기본 Auto Wiring
  + <beans> 엘리먼트 내부에 default-autowire=”~~~” 로 지정하여 주어진 설정파일 내의 모든 빈의 기본 Auto Wiring설정으로 사용한다.
* 명시적 와이어링과 혼용
  + autowire설정과 property 엘리먼트는 혼용함으로써 byType에서 가능한 모호함을 방지할 수 있다. 그러나 constructor 오토와이어링시에는 <constructor-arg>엘리먼트는 사용할 수 없다.
* 애너테이션을 이용한 와이어링
  + <beans> </beans> 내부에 <context:annotation-config/> 엘리먼트 선언
  + @Autowired 애너네이션
    - 빈 레퍼런스에 byType 와이어링 하기 위해 해당되는 메소드에 애너테이션 사용
    - 생성자와 함게 사용하면 xml의 <constructor-arg> 엘리먼트 생략 가능
    - 프로퍼티를 직접 애너테이션 / 세터를 사용할 필요가 없다.
      * @Autowired  
        private Instrument instrument; -> private도 사용 가능함
  + 선택적 오토와이어링
    - @Autowired(required=false) -> Instrument 타입의 빈을 찾지 못하면 null 프로퍼티  
      private Instrument instrument;
  + 모호한 종속개체 제한
    - 해당되는 빈이 두 개 이상일 경우 필요함
    - @Autowired  
      @Qualifier(“guitar”)   
      private Instrument instrument;
    - ID가 guitar인 빈에 와이어링 한다.
    - 또는 XML에 qualifier 값을 지정할 수 있음.  
      <bean class=”com.~.Guitar”>  
       <qualifier value=”stringed”/>  
      </bean>  
        
      @Qualifier(“stringed”)  
      public class Guitar implements Instrument  
      {..}
  + 커스텀 한정자 생성
    - 한정자 생성  
      package com.springinaction.springidol.qualifiers;  
      import 생략  
        
      @Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})  
      @Retention(RetentionPolicy.RUNTIME)  
      @Qualifier  
      public @interface StringedInstrumnet{ }
    - 빈에 적용  
      @StringedInstrument  
      public class Guitar implements Instrument  
      {…}
    - 오토와이어링시  
      @Autowired  
      @StringedInstrument  
      private Instrument instrument;
    - 한정자는 여러 개를 겹쳐서 세밀한 지정이 가능하다  
      @Autowired  
      @StringedInstrument -> 첫번째 커스텀 한정자  
      @Strummed -> 두번째 커스텀 한정자  
      private Instrument instrument’
* @Inject를 이용한 표준 기반의 오토 와이어링
  + 배경 : DI프레임워크의 모델 통합을 위한 자바 명세 JSR-330 : 자바의 공통 DI 모델
  + @Autowired를 거의 완벽히 대체  
    @Inject  
    private Instrument instrument;
    - 공통점 : 프로퍼티, 메소드,, 생성자 오토와이어링 사용
    - 차이점 : @Inject는 required애트리뷰가 없음
  + Provider 인터페이스로 빈 레퍼런스의 lazy injection과 빈의 멀티 인스턴스 주입  
    (Knife빈의 scope를 prototype으로 선언했다고 가정하고)  
      
    private Set<Knife> knives;  
      
    @Inject  
    public KnifeJuggler(Provider<Knife> knifeProvider){  
     knives = new HashSet<Knife>();  
     for (int i=0; i<5;i++){  
     knives.add(knifeProvider.get());  
     }  
    }  
    생성자에 knife 인스턴스 대신에 Provider<Knife>를 받고,  
    실제 knife 인스턴스는 get()메소드 호출될때마다 주입된다.
  + @Qualifier = @named  
      
    @Inject  
    @Named(“guitar”)  
    private Instrument instrument;  
      
    qualifier 는 빈의 ID들로 선택범위를 좁히고, Named는 구체적인 id에 의한 빈 선택을 한다.
  + 커스텀 JSR-330 한정자 생성
    - 스프링의 생성방식과 동일하며 import문에서만 차이가 있다.   
        
      동일 : import java.lang.annotation;   
      스프링 : import org.springframework.beans.factory.annotation;  
      JSR-330 : import javax.inject.Qualifier;
* 애너테이션을 이용한 int, Boolean, String값 와이어링
  + @Value(“Eruption”)  
    private String song;  
    -> 이런 단순한 주입은 하드코딩이 더 적합.
  + SpEL 표현식을 통한 @Value 사용
    - @Value(“#{systemProperties.myFavoriteSong}”)  
      private String song;
* 자동으로 빈 발견하기
  + <context : annotation-config> : <bean>을 이용하여 빈을 명시적 선언해야함

<context:component-scan base-package=”com.~”> : 상기 기능을 포함하여, 자동으로 빈을 발견/선언하게 한다. base-package는 빈을 검색할 패키지

* + 오토디스커버리를 위한 빈 애너테이션 (클래스 위에 정의)
    - @Component : 범용 스테레오타입 애너테이션
    - @Controller : 스프링 MVC 컨트롤러
    - @Repository : 데이터 저장소
    - @Service : 서비스를 정의함
  + 기본적으로 빈의 id는 클래스이름의 카멜 케이싱 이름으로 Guitar 클래스는 id가 guitar임.  
    별도로 id 명시하기 위해서는 @Component(“eddie”) 처럼 지정한다.
  + 컴포넌트 스캔 필터링
    - 빈 등록할 클래스마다 애너테이션을 기재하기 번거로울 경우,   
      빈 등록하거나 / 등록하지 않을 검색 조건을 지정할 수 있다.
    - <context:component-scna  
       base-package=”com.springinaction.springidol”>  
       <context:include-filter type=”assignable”  
       expression=”com.springinaction.springidol.Instrument”/>  
       <context:exclude-filter type=”annotation”  
       expression=”com.springinaction.springidol.SkipIt”/> @SkipIt 적용 클래스는 생략  
      </context:component-scan>
    - 필터의 종류
      * annotation : 클래스에 지정한 애너테이션이 적용됐는지의 여부
      * assignable : 클래스가 지정한 타입으로 할당 가능한지의 여부
      * aspect : 클래스 이름이 AspectJ 표현식에 일치하는지의 여부
      * custom : 커스텀 구현체 사용
      * regex : 클래스 이름이 정규 표현식에 일치하는지의 여부
* 스프링의 자바 기반 설정 사용
  + 아래의 최소한의 XML은 필요함  
    <beans xmlns=~~~~>  
     <context:component-scan base-package=”~~~”/>  
    </beans>
  + 그 외의 설정은 scan 대상 클래스 중 @Configuration 애너테이션이 적용된 클래스 사용
    - 컴파일 시점에 몇몇 설정은 체크 가능하다는 이점이 있다.
    - @Configuration  
      public class SpringIdolConfig{  
        
       @Bean  
       public Perfomer duke(){  
       return new Juggler();  
       } -> duke ID로 Juggler 인스턴스를 스프링 빈 등록  
        
       @Bean  
       public Perfomer duke15(){  
       return new Juggler(15);  
       } -> <constructor-arg> 역할 수행  
        
       @Bean  
       public Performer kenny(){  
       Instrumentalist Kenny = new Instrumentalist();  
       Kenny.setSong(“Jingle Bells”); -> 세터 주입  
       return Kenny;  
       }  
        
      @Bean  
       public Perfomer poeticDuke(){  
       return new PoeticJuggler(sonnet29());   
       } -> sonnet29() 메소드에 @Bean 애너테이션이 적용되어 있다면, 새로운 인스턴스가 아닌 컨텍스트에 등록된 빈을 가져온다.  
        
      }

4장 애스펙트 지향 스프링

* Aspect-Oritented Programming의 목적 : 횡단관심사와 이에 영향받는 객체간 결합도를 낮춤.  
  적용사례 : 로깅 / 선언적 트랜잭션, 보안, 캐싱
* AOP 용어 정리
  + 어드바이스 : 애스펙트가 언제 무엇을 해야할지 정의
    - 언제 : 메소드의 실행 이전 / 이후 / 반환 이후(성공적 완료) / 예외발생 이후 / 전후
  + 조인포인트 : 어드바이스를 적용할 수 있는 곳
  + 포인트컷 : 어드바이스를 적용할 곳 (조인포인트 중 하나)
  + 애스펙트 : 어드바이스 + 포인트컷
  + 인트로덕션 : 기존 클래스 변경 없이 새 메소드나 멤버 변수를 추가하는 기능
  + 위빙 : 타깃 객체에 애스펙트를 적용해서 새로운 프록시 객체를 생성하는 절차
    - * 위빙의 시점 : 컴파일시간 / 클래스로드시간 / 실행시간
* 스프링 애스펙트의 실행
  + 실행시 프록시 객체(스프링 빈에 애스펙트가 위빙) 생성
  + 호출시 프록시 객체가 타깃객체로 위장하여 메소드의 호출을 가로채고, 타깃 객체로 호출전달
  + 컨테이너가 모든 빈을 생성한 이후에 프록시 타깃 객체를 생성
  + 스프링 AOP에서는 프록시가 실행시간에 생성되므로 별도 애스펙트 위빙 컴파일러 필요치 않음.
  + 스프링 AOP는 메소드 조인포인트만 가능. (AspectJ나 JBoss에서는 필드/생성자 조인포인트 가능)
* 포인트컷을 이용한 조인포인트 선택
  + 스프링은 AspectJ의 포인트컷 표현식 언어를 사용함
  + 포인트컷 표현식 언어
    - args() 인자가 주어진 타입의 인스턴싄 조인포인트를 정의
    - @args() 전달된 인자의 런타임 타입이 주어진 타입의 애너테이션을 갖는 조인포인트 정의
    - execution() 메소드 실행 조인퐁인트와 일치시키는데 사용
    - this() 빈 레퍼런스가 주어진 타입의 인스턴스를 갖는 조인포인트를 정의한다
    - target() 대상 객체가 주어진 타입을 갖는 조인포인트를 정의
    - @target() 수행 중인 객체의 클래스가 주어진 타입의 애너테이션을 갖는 조인포인트를 정의
    - within() 특정 타입에 속하는 조인포인트를 정의
    - @within() 주어진 애너테이션을 갖는 타입 내 조인포인트를 정의
    - @annotation 조인포인트의 대상 객체가 주어진 애너테이션을 갖는 조인포인트를 정의  
        
      \* execution 지정자만 일치 작업(기본 지정자) / 다른 지정자는 일치 제한
  + 포인트컷 작성
    - execution( \* com.springinaction.springidol.Instrument.play(…))
      * \* : 메소드의 반환 타입 무엇이든 상관없음
      * Instrument 클래스의 play 메소드 실행
      * (..) : 메소드 인자 목록이 무엇이든 상관없음
    - execution( \* com.springinaction.springidol.Instrument.play(…))   
       && within(com.springinaction.springidol.\*)
      * 메소드가 com.~springidol 패키지의 클래스 안에서 호출되는 경우로 제한
      * && : and 조건
      * || : or 조건
      * ! : 지정자의 부정
    - execution( \* com.springinaction.springidol.Instrument.play()) and bean(eddie)
      * Instrument의 play 메소드 중 ID가 eddie인 빈