**Spring**

1. **스프링 개발 툴**

**#자바 빌드툴 - Maven, Gradle**

* JAR 의존체 (Dependency 다운로드), 컴파일에 필요한 클래스패스 설정, JAR 구성 파일 복사)

**#그레이들 프로젝트 임폴트 및 빌드**

1. <http://www.gradle.org/downloads> 에서 gradle 다운로드.
2. 환경변수에 **gradle /bin** 디렉토리 추가.
3. **gradle wrapper –gradle-version** 현재버전 명령어로 **gradlew** 설치
4. **gradlew build 명령어** 로 프로젝트 빌드. gradlew 은 gradle 버전의 호환 문제를 해결한다.
5. **build/libs** 폴더에 실행 가능한 **jar 파일**이 생성된다.
6. **Java -jar [jar파일]** 로 빌드된 어플리케이션 실행.
7. **스프링 코어**

**IoC (Inversion of Control) Container** 는 **POJO(Plain Old Java Object)** 를 구성하고 관리한다.

스프링은 **POJO** 와 **IoC 컨테이너를 다루는 기술**이다.

Container에서 사용할 컴포넌트를 생성하기 위해선, POJO 클래스를 설계한 후 **@Configuration, @Bean** 을 붙인 **구성 클래스**를 생성하거나 **@Component, @Repository, @Service, @Controller** 등을 붙인 **자바 Component**을 생성한다.

**IoC 컨테이너**는 위의 어노테이션 클래스를 붙인 자바 클래스를 **Scanning** 하여 어플리케이션의 구성 요소로 사용한다.

**#구성 클래스**

**@Configuration** 은 해당 클래스를 구성 클래스로 구성한다.

구성 클래스안의 **@Bean 메서드**는 **Bean 인스턴스**를 생성후 반환한다.

@Configuration  
public class GeneralConfiguration {  
 // 해당 빈의 이름은 seqGen (메서드 이름) 이다.  
 @Bean  
 public SequenceGenerator seqGen() {  
 SequenceGenerator gen = new SequenceGenerator();  
 gen.setPrefix("30");  
 gen.setSuffix("A");  
 gen.setInitial(1000000);  
 return gen;  
 }  
  
}

**#IoC 컨테이너에서 POJO/BEAN 가져오기.**

스프링은 **기본 구현체 Bean Factory** 와 **고급 구현체인 컨텍스트 Application Context**, 두 가지 **IoC 컨테이너**를 제공한다. **빈 인스턴스**를 가져오기 위해선 **IoC Container** 를 **Instantiation** 해야 한다.

**ApplicationContext** 은 POJO 인스턴스 또는 **빈(Bean)**에 Access 하는 **창구 역할**을 한다.

**ApplicationContext** 에서 구성 클래스에 선언된 빈은 **getBean() 메서드**를 호출하여 가져올 수 있다.

@Test  
public void applicationContext() {  
 ApplicationContext context = new AnnotationConfigApplicationContext(GeneralConfiguration.class);

// 빈의 이름과 타입(Optional)을 명시해준다.  
 SequenceGenerator gen = context.**getBean**("seqGen", SequenceGenerator.class);  
 *assertThat*(gen).isNotNull();  
}

**#POJO 클래스에 @Component를 붙여 빈 선언과 생성.**

**POJO** 클래스에 **@Component** 를 붙이면 스프링은 이 클래스를 이용해 Bean 을 생성한다**. @Component 는** POJO 에 붙이는 **범용 어노테이션**이며, 각 **계층 영속화(Persistence), 서비스(Service), 표현(Controller)**, 세 레이어(Layer)에 맞는 **@Repository, @Service, @Controller** 어노테이션을 지원한다.특정 레이어에 맞는 부가 기능을 맞출려면 레이어에 맞는 어노테이션을 사용하는 것이 좋다**.** (Ex: **@Repository**는 발생한 예외를 **DataAccessException** 으로 감싸 던진다.)

// 해당 빈의 이름은sequenceDao이다.

@Component("sequenceDao")  
public class SequenceDaoImpl implements SequenceDao {  
  
 private final Map<String, Sequence> sequences = new HashMap<>();  
 private final Map<String, AtomicInteger> values = new HashMap<>();  
  
 public static final String *IT* = "IT";  
 public static final String *IT\_PREFIX* = "30";  
 public static final String *IT\_SUFFIX* = "A";  
 public static final int *IT\_INITIAL\_VALUE* = 10000;  
  
 public SequenceDaoImpl() {  
 sequences.put(*IT*, new Sequence(*IT*, *IT\_PREFIX*, *IT\_SUFFIX*));  
 values.put(*IT*, new AtomicInteger(*IT\_INITIAL\_VALUE*));  
 }  
  
 @Override  
 public Sequence getSequence(String sequenceId) {  
 return sequences.get(sequenceId);  
 }  
  
 @Override  
 public int getNextValue(String sequenceId) {  
 AtomicInteger value = values.get(sequenceId);  
 return value.getAndIncrement();  
 }  
}

**#어노테이션 Scanning 과 Filter**

하나 이상의 포함/제외 필터를 적용해서 스캐닝 (Scanning) 과정을 Customize 할 수 있다.

* 필터링할때, **basePackages**로 스캐닝의 디폴트 패스를 지정해야 한다.

**스프링이 지원하는 필터 표현식은 annotation, assignable, regex, aspectj 이다.**

**@ComponentScan** 의 **includeFilters** 은 검색 대상을 포함할 필터, **excludeFilters** 은 검색 대상을 제외할 필터이다.

@Configuration  
@ComponentScan(basePackages = "com.siwoo",  
 includeFilters = @ComponentScan.Filter(  
 type = FilterType.*REGEX*,  
 pattern = {"com.\*dao.\*Dao.\*"}  
 ),  
 excludeFilters = @ComponentScan.Filter(  
 type = FilterType.*REGEX*,  
 pattern = "com.\*.dao\\.MyBatis.\*DaoImpl"  
 ))  
public class GeneralConfiguration {  
  
 @Bean  
 public SequenceGenerator seqGen() {  
 SequenceGenerator gen = new SequenceGenerator();  
 gen.setPrefix("30");  
 gen.setSuffix("A");  
 gen.setInitial(1000000);  
 return gen;  
 }  
  
}

@Test  
public void testScanningFilter() {  
 final String target = "myBatisSequenceDaoImpl";  
 ApplicationContext c = new AnnotationConfigApplicationContext(GeneralConfiguration.class);  
 for (String bean: c.getBeanDefinitionNames()) {  
 if (bean.equalsIgnoreCase(target)) {  
 *fail*();  
 }  
 }  
}

**#POJO 에서 다른 컴포넌트와 상호작용.**

어플리케이션은 어플리케이션을 구성하는 컴포넌트와 상호작용하여 임무를 완수한다.

1. **구성 클래스에서 의존성 연결**

**@Bean 어노테이션**으로 빈을 정의한 후, 빈 메서드를 호출하여 의존성을 연결시켜준다.

@Bean  
public DatePrefixGenerator datePrefixGenerator() {  
 DatePrefixGenerator prefixGen = new DatePrefixGenerator("yyyyMMdd");  
 return prefixGen;  
}  
  
@Bean  
public SequenceGenerator seqGen() {  
 SequenceGenerator gen = new SequenceGenerator();

//Injecting dependency  
 gen.setPrefix(datePrefixGenerator());  
 gen.setSuffix("A");  
 gen.setInitial(1000000);  
 return gen;  
}

1. **POJO 필드에 @Autowired을 사용하여 의존성 연결.**

**@Component** 로 등록된 빈에서 **@Autowired** 를 프로퍼티에 적용하면 레퍼런스가 자동으로 연결된다 (**타입**으로 의존하는 빈을 찾아온다).

* **배열 프러퍼티에** **@Autowired**을 붙이면 매칭된 빈을 모두 찾아 자동 연결시킨다.
* **Map 프러퍼티에 @Autowired**을 붙이면 키를 **Bean Name,** 값을 **Bean 레퍼런스로** 매핑시켜 연결시킨다.
* @Service  
  public class SequenceService {  
    
   @Autowired  
   private SequenceDao sequenceDao;  
    
   @Autowired  
   private PrefixGenerator[] prefixGens;  
   @Autowired  
   private Map<String, PrefixGenerator> prefixGensMap;  
    
   public PrefixGenerator[] getPrefixGens() {  
   return Arrays.*copyOf*(prefixGens, prefixGens.length);  
   }  
    
   public Map<String, PrefixGenerator> getPrefixGensMap() {  
   return new HashMap<>(prefixGensMap);  
   }  
    
   public void setSequenceDao(SequenceDao sequenceDao) {  
   this.sequenceDao = sequenceDao;  
   }  
    
   public String generate(String sequenceId) {  
   Sequence seq = sequenceDao.getSequence(sequenceId);  
   int value = sequenceDao.getNextValue(sequenceId);  
   return seq.getPrefix() + value + seq.getSuffix();  
   }  
  }
* @Test  
  public void testSequenceService() {  
   SequenceService seqService = sequenceContext.getBean(SequenceService.class);  
   *assertThat*(seqService.generate("IT")).isNotNull();  
   *assertThat*(seqService.getPrefixGens().length).isEqualTo(2);  
   for (PrefixGenerator prefixGen: seqService.getPrefixGens()) {  
   String beanName = sequenceContext.getBeanNamesForType(prefixGen.getClass())[0];  
   *assertThat*(beanName).isNotBlank();  
   PrefixGenerator resolvedGen = seqService.getPrefixGensMap().get(beanName);  
   System.*out*.println(resolvedGen.getPrefix());  
   }  
  }

**#@Autowired 와 선택적 연결**

스프링은 **@Autowired** 로 의존 연결할 빈을 찾지 못하면 예외를 던진다. 대신, **@Autowired 의 required 속성값**을 **false** 로 지정하면 예외를 던지지 않는다.

public class SequenceGenerator {  
 private PrefixGenerator prefixGen;  
 private String suffix;  
 private int initial;  
 private final AtomicInteger counter = new AtomicInteger();  
  
 public SequenceGenerator() {  
 }  
  
 public String getSequence() {  
 StringBuilder sb = new StringBuilder();  
 sb.append(prefixGen.getPrefix())  
 .append(initial)  
 .append(counter.getAndIncrement())  
 .append(suffix == null ? "" : suffix);  
 return sb.toString();  
 }  
  
 @Autowired(required = false)  
 public void setPrefix(DatePrefixGenerator prefixGen) {  
 if (prefixGen == null)  
 this.prefixGen = new SimplePrefixGenerator();  
 else  
 this.prefixGen = prefixGen;  
 }  
  
 public void setSuffix(String suffix) {  
 this.suffix = suffix;  
 }  
  
 public void setInitial(int initial) {  
 this.initial = initial;  
 }  
}

**#모호한 의존성 자동 연결 Resolving**

**타입을 기준으로 자동 연결할(Autowiring)** 시, IoC 컨테이너에 정의된 호환 타입이 여럿 존재일 경우 예외가 발생한다. 이럴때 **@Primary, @Qualifier** 로 **의존 연결 충돌**을 해결할 수 있다.

**@Qualifier** 대신 **JSR-250의 @Resource** 을 이용하여 빈 이름을 주어 충돌을 해결할 수 있다.

@Component

// 해당 빈에게 우선권을 부여한다.

@Primary  
public class DatePrefixGenerator implements PrefixGenerator {  
  
 private String pattern;  
 private DateTimeFormatter formatter;  
  
 public DatePrefixGenerator(@Value("yyyyMMdd") String pattern) {  
 this.pattern = pattern;  
 this.formatter = DateTimeFormatter.*ofPattern*(pattern);  
 }  
  
 public void setPattern(String pattern) {  
 this.pattern = pattern;  
 }  
  
 public String getPrefix() {  
 return LocalDate.*now*().format(formatter == null ? DateTimeFormatter.*ISO\_DATE* : formatter);  
 }  
}

@Service  
public class SequenceService {  
  
 @Autowired  
 private SequenceDao sequenceDao;  
  
 @Autowired

// 타입 의존 연결하기 위해 bean name 을 매칭시키지 않는다.  
 private PrefixGenerator prefixGenerator;  
  
 @Autowired  
 private PrefixGenerator[] prefixGens;  
  
 @Autowired  
 private Map<String, PrefixGenerator> prefixGensMap;  
  
 public PrefixGenerator getPrefixGenerator() {  
 return prefixGenerator;  
 }  
  
 public PrefixGenerator[] getPrefixGens() {  
 return Arrays.*copyOf*(prefixGens, prefixGens.length);  
 }  
  
 public Map<String, PrefixGenerator> getPrefixGensMap() {  
 return new HashMap<>(prefixGensMap);  
 }  
  
 public void setSequenceDao(SequenceDao sequenceDao) {  
 this.sequenceDao = sequenceDao;  
 }

@Test  
public void testSequenceService() {  
 SequenceService seqService = sequenceContext.getBean(SequenceService.class);  
 *assertThat*(seqService.generate("IT")).isNotNull();  
 *assertThat*(seqService.getPrefixGenerator()).isInstanceOf(DatePrefixGenerator.class);  
 //assertThat(seqService.getPrefixGens().length).isEqualTo(2);  
 for (PrefixGenerator prefixGen: seqService.getPrefixGens()) {  
 String beanName = sequenceContext.getBeanNamesForType(prefixGen.getClass())[0];  
 *assertThat*(beanName).isNotBlank();  
 PrefixGenerator resolvedGen = seqService.getPrefixGensMap().get(beanName);  
 System.*out*.println(resolvedGen.getPrefix());  
 }  
}

**#분산된 구성 클래스의 POJO 참조**

어플리케이션의 규모가 커질수록 모든 POJO 설정을 하나의 자바 구성 클래스에 담아두기 어렵기 때문에 기능에 따라 여러 자바 구성 클래스로 나누어 관리한다. 여럿 구성 클래스가 공존할시, **ApplicationContext 를 초기화할 때 경로를 명시**하거나**, @Import** 구성 파일을 나누어 임폴트할 수 있다.

다른 구성 클래스의 빈의 의존체가 필요할 시, **@Value 와 SpEL** 혹은 **@Autowired** 을 이용하여 레퍼런스를 가져올 수 있다.

@Import({ShopConfiguration.class, GeneralConfiguration.class})  
public class RootConfiguration {  
  
}

@Configuration  
@ComponentScan(basePackages = "com.siwoo.chapter2.shop")  
public class ShopConfiguration {  
  
 @Value("#{sequenceService}")  
 private SequenceService seqService;  
  
 @Bean  
 public Product aaa() {  
 Battery aaa = new Battery("AAA", 2.5, seqService.generate("IT"));  
 aaa.setRechargeable(true);  
 return aaa;  
 }  
  
 @Bean  
 public Product cdrw() {  
 Disc p2 = new Disc("CD-RW", 1.5, seqService.generate("IT"));  
 p2.setCapacity(700);  
 return p2;  
 }  
}

@Test  
public void initContext() {  
 ApplicationContext c = new AnnotationConfigApplicationContext(GeneralConfiguration.class, ShopConfiguration.class);  
 *assertThat*(c.getBean("aaa")).isNotNull();  
 *assertThat*(c.getBean("sequenceService")).isNotNull();  
 System.*out*.println(c.getBean("sequenceService", SequenceService.class).generate("IT"));  
}  
  
@Test  
public void usingImport() {  
 ApplicationContext c = new AnnotationConfigApplicationContext(RootConfiguration.class);  
 *assertThat*(c.getBean("aaa")).isNotNull();  
 *assertThat*(c.getBean("sequenceService")).isNotNull();  
 System.*out*.println(c.getBean("sequenceService", SequenceService.class).generate("IT"));  
}

**#@Autowired** 대신 **자바 표준 어너테이션 @Resource, @Inject**을 이용한 의존성 연결.

**@Resource** 는 **JSR-250(Common Annotations for the Java Platform)** 에 규정된 어노테이션으로, **이름**으로 POJO 레퍼런스를 찾아 의존 연결한다. **JSR-330(standard Annotation for Injection: javax.inject)** 에 규정된 어노테이션으로 **타입**으로 POJO 레퍼런스를 찾아 의존 연결한다.

**#커스텀 어노테이션을 이용한 의존 연결.**

[**Javax.inject.@Qualifier**](mailto:Javax.inject.@Qualifier) **을 이용하여 어노테이션을 정의한 후 해당 클래스를 빈으로 설정한다.**

@Qualifier  
@Target({*TYPE*, *FIELD*, *PARAMETER*})

**//문서화시에도 해당 어노테이션의 정보를 표기한다.**  
@Documented  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface PrefixBean {  
  
}

@PrefixBean  
@Primary  
public class DatePrefixGenerator implements PrefixGenerator {

**#@Scope 을 사용하여 POJO 의 생명주기 지정.**

**@Scope** 는 빈Bean 스코프를 지정한다. **Singleton** 은 기본 스코프로써 IoC 는 하나의 인스턴스만을 생성하여 공유한다. **Prototype** 은 Bean 을 요청할 때마다 새로운 인스턴스를 새로 만든다. **Request** 는 HTTP 요청당 하나의 빈 인스턴스를 생성한다 (**Web Application Context** 에게 요청). **Session** 은 HTTP 세션당 빈 인스턴스 하나를 생성한다. (**Web Application Context** 에게 요청).

@Bean  
@Primary  
public ShoppingCart shoppingCart() {  
 return new ShoppingCart();  
}  
  
@Bean  
@Scope("prototype")  
public ShoppingCart protoShoppingCart() {  
 return new ShoppingCart();  
}

@Test  
public void testPrototype() {  
 ShoppingCart shoppingCart = c.getBean(ShoppingCart.class);  
 Product aaa = c.getBean("aaa", Product.class);  
 Product cdrw = c.getBean("cdrw", Product.class);  
 Product dvdrw = c.getBean("dvdrw", Product.class);  
 shoppingCart.addItem(aaa);  
 shoppingCart.addItem(cdrw);  
 shoppingCart = c.getBean(ShoppingCart.class);  
 shoppingCart.addItem(dvdrw);  
 *assertThat*(shoppingCart.getItems()).contains(aaa, cdrw, dvdrw);  
  
 ShoppingCart protoCart = c.getBean("protoShoppingCart", ShoppingCart.class);  
 protoCart.addItem(aaa);  
 protoCart.addItem(cdrw);  
 *assertThat*(protoCart.getItems()).contains(aaa, cdrw);  
 protoCart = c.getBean(ShoppingCart.class);  
 protoCart.addItem(dvdrw);  
 *assertThat*(protoCart.getItems()).contains(dvdrw);  
}

**#외부 리소스(TEXT, XML, Property, Image) 가져오기**

**@PropertySource**을 이용하면 **Properties(key-value)** 파일을 읽어들일 수 있다. 또한 **org.springframework.core.io.Resource** 라는 단일 인터페이스를 사용해 어떤 유형의 외부 리소스를 **같은 매커니즘으로 로드**할 수 있다.

**@Value** 어노테이션을 통해 특정 위치에 존재하는 리소스를 불러올 수 있으며, 특정 위치에 맞는 표현식 **file[로컬 파일시스템]**, **classpath[클래스패스]** 접두어를 사용하여 편리하게 로드할 수 있다.

**#Properties 파일 로드하기**

**@PropertySource 와 PropertySourcesPlaceholderConfigurer (Spring Boot 에서는 Optional)** 클래스. **@Value 어노테이션 과 표현식(${})**를 이용해 프러퍼티 설정 파일을 로드한다.

@Configuration  
@PropertySource("classpath:/app/discounts.properties")  
@ComponentScan(basePackages = "com.siwoo.chapter2.shop")  
public class ShopConfiguration {  
  
// @Value("#{sequenceService}")  
 @Resource(name = "sequenceService")  
 private SequenceService seqService;

//Placeholder expression  
 @Value("${special.discount}")  
 private double SPECIAL\_DISCOUNT;

// *Specialization of {****@link*** *PlaceholderConfigurerSupport} that resolves ${...} placeholders  
 \* within bean definition property values and {****@code*** *@Value} annotations*  
 @Bean  
 public static PropertySourcesPlaceholderConfigurer  
 propertySourcesPlaceholderConfigurer() {  
 return new PropertySourcesPlaceholderConfigurer();  
 }  
  
 @Bean  
 public Product aaa() {  
 Battery aaa = new Battery("AAA", 2.5, seqService.generate("IT"));  
 aaa.setRechargeable(true);  
 return aaa;  
 }  
  
 @Bean  
 public Product cdrw() {  
 Disc p2 = new Disc("CD-RW", 1.5, seqService.generate("IT"), SPECIAL\_DISCOUNT);  
 p2.setCapacity(700);  
 return p2;  
 }  
  
 @Bean  
 public Product dvdrw() {  
 Disc p2 = new Disc("DVD-RW", 3.0, seqService.generate("IT"), -1);  
 p2.setCapacity(850);  
 return p2;  
 }

public class Disc extends Product {  
  
 private int capacity;  
 private double discount;  
 public static final double *PLACES* = Math.*pow*(10, 2);  
  
 public Disc() {  
 }  
  
 public Disc(String name, double price, String id, double discount) {  
 super(name, price, id);  
 this.discount = discount;  
 if (discount > 0.0D) {  
 price = Math.*round*((price - (price \* discount)) \* *PLACES*) / *PLACES*;  
 setPrice(price);  
 }  
 }

**#외부 리소스 파일 데이터 로드하기**

**Resource 필드**를 선언하고 **@Value 어노테이션**을 이용해 **파일 경로**를 지정하면 해당 프러퍼티에 **Resource 객체의 레퍼런스를 주입**한다.

* **파일 경로 접두어**

**classpath:** 해당 파일의 시작점을 클래스 패스로 지정한다.

**http:** 해당 파일을 URL 의 위치로 지정한다.

**file:** 해당 파일의 시작점을 파일 시스템 패스로 지정한다.

* **PropertiesLoaderUtils**

입력 스트림의 표준 처리를 수행하여 **Properties 로드** 시 편리한 유틸리티 클래스.

//FileSystem 경로

[//@Value("file:C:/Users/Siwoo/IdeaProjects/GitHub/chapter2/src/main/resources/app/banner.txt")](mailto://@Value(%22file:C:/Users/Siwoo/IdeaProjects/GitHub/chapter2/src/main/resources/app/banner.txt%22))

//Classpath 경로  
@Value("classpath:/app/banner.txt")  
private org.springframework.core.io.Resource resource;  
  
@Bean  
public BannerLoader bannerLoader() {  
 BannerLoader banner = new BannerLoader(resource);  
 return banner;  
}

@Test  
public void banner() throws IOException {  
 BannerLoader banner = c.getBean(BannerLoader.class);  
 banner.showBanner();  
 //Resource resource = new ClassPathResource("app/discount.properties");  
 Resource resource = new FileSystemResource(Paths.*get*(".", "src", "main", "resources", "app", "discounts.properties"));  
 Properties props = **PropertiesLoaderUtils.*loadProperties*(resource);** *assertThat*(props.getProperty("special.discount")).isEqualTo(Double.*toString*(0.1D));  
}

**#i18n 메세지 다국적화**

**org.springframework.context.MessageSource 인터페이스** 는 **리소스 번들 메시지(Resource Bundle Message)**를 처리하는 메서드가 정의되어 있다. **ReloadableResourceBundleMessageSource** 는 **MessageSource 의 구현체**로 **Locale** 별로 분리된 리소스 번들 메세지를 해석한다. **ResourceBundleMessageSource** 의 **baseNames** 는 번들의 위치를 설정하며, **cacheSeconds** 은 해당 메세지를 캐쉬할 시간을 지정한다.

메세지를 런타임에 로드할때는**MessageSource 인터페이스**를 의존 주입받아 사용하면 된다.

* **Resource Bundle Properties** 의 이름 명명규칙.

**[bundle\_name]\_[Locale].properties.**

Ex) messages\_en.properties / messages\_**en\_US**.properties

Ex) messages\_ko.properties / messages\_**ko\_KR**.properties

@Configuration  
@PropertySource("classpath:/app/discounts.properties")  
@ComponentScan(basePackages = "com.siwoo.chapter2.shop")  
public class ShopConfiguration {  
 private org.springframework.core.io.Resource resource;  
  
 @Bean  
 public **ReloadableResourceBundleMessageSource** messageSource() {  
 ReloadableResourceBundleMessageSource message = new ReloadableResourceBundleMessageSource();  
 **message.setDefaultEncoding(Charset.*forName*("UTF-8").name());  
 message.setBasenames("classpath:/app/messages");**  
 **message.setCacheSeconds(1);**  
 return message;  
 }

@Test  
public void testMessage() {  
 Disc disc = c.getBean("cdrw", Disc.class);  
 //ko test  
 String message = c.getMessage("alert.checkout", null, Locale.*getDefault*());  
 *assertThat*(message).isEqualTo("쇼핑 카트가 대출되었습니다.");  
 String time = LocalDate.*now*().toString();  
 message = c.getMessage("alert.inventory.checkout", new String[]{disc.getName(),time}, Locale.*getDefault*());  
 *assertThat*(message).isEqualTo(String.*format*("%s이(가) 있는 쇼핑 카트가 %s에 체크아웃되었습니다.", disc.getName(), time));  
 //en test  
 message = c.getMessage("alert.checkout", null, Locale.*ENGLISH*);  
 *assertThat*(message).isEqualTo("A shopping cart has been checked out.");  
 time = LocalDate.*now*().toString();  
 message = c.getMessage("alert.inventory.checkout", new String[]{disc.getName(),time}, Locale.*ENGLISH*);  
 *assertThat*(message).isEqualTo(String.*format*("A shopping cart with %s has been checked out at %s", disc.getName(), time));  
}

@Component  
public class Cashier {  
  
 @Autowired  
 private MessageSource message;  
  
 public void checkout(ShoppingCart cart) {  
 String alert = message.getMessage("alert.inventory.checkout",  
 new Object[]{ cart.getItems(), new Date()}, Locale.*US*);  
 System.*out*.println(alert);  
 }  
  
}

**#빈의 (POJO) 초기화/폐기 관리**

**IoC 컨테이너**에서 **빈을 초기화 및 폐기하는 로직**을 커스트마이징 할 수 있다. **@Bean** 정의시 **initMethod**, **destroyMethod** 속성을 통해 초기화, 폐기 콜백 메서드를 지정한다. 또 **@Lazy** **은 느긋한 초기화(Lazy initialization)** 기법을 사용할 수 있고, **@DependsOn** 을 통해 빈을 생성하기 전 다른 빈을 먼저 생성하도록 강제할 수 있다.

@Bean(**initMethod** = "initResource", **destroyMethod** = "closeResource")  
public Cashier cashier() {  
 String path = System.*getProperty*("java.io.tmpdir") + "cashier";  
 Cashier cashier = new Cashier(path, "checkout");  
 return cashier;  
}

public class Cashier {  
  
 @Autowired  
 private MessageSource message;  
  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(Cashier.class);  
  
 private Writer writer;  
 private String path;  
 private String fileName;  
  
 public Cashier(String path, String fileName) {  
 this.path = path;  
 this.fileName = fileName;  
 }

//Init Callback  
 private void initResource() {  
 try {  
 System.*out*.println(path);  
 File dir = new File(path);  
 if (!dir.exists())  
 dir.mkdir();  
 File file = new File(dir, fileName);  
 if (!file.exists())  
 file.createNewFile();  
 writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file, true)));  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in creating Cashier file.", e);  
 throw new CashierResourceException("Error's in creating Cashier file.", e);  
 }  
 }

//Destroy Callback  
 private void closeResource() {  
 try {  
 writer.close();  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in closing Cashier file.", e);  
 throw new CashierResourceException("Error's in closing Cashier file.", e);  
 }  
 }  
  
 public void checkout(ShoppingCart cart) {  
 String alert = message.getMessage("alert.inventory.checkout",  
 new Object[]{ cart.getItems(), new Date()}, Locale.*US*);  
 try {  
 writer.write(alert);  
 writer.flush();  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in writing on Cashier file.", e);  
 throw new CashierResourceException("Error's in writing on Cashier file.", e);  
 }  
 }  
  
 static class CashierResourceException extends RuntimeException {  
 public CashierResourceException(String message, Exception e) {  
 super(message, e);  
 }  
 }  
  
}

**#@PostConstruct와 @PredEstroy로 POJO 초기화/폐기 메서드 지정하기.**

자바 구성 클래스가 아닌 POJO 클래스에 직접 **@PostContruct, @PreDestroy** 를 이용해 초기화/ㅍ{기 메서드를 지정할 수 있다.

public class Cashier {  
  
 @Autowired  
 private MessageSource message;  
  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(Cashier.class);  
  
 private Writer writer;  
 @Value("classpath:/app")  
 private Path path;  
 @Value("checkout")  
 private String fileName;  
  
 public Cashier() {  
 }  
  
 @PostConstruct  
 private void initResource() {  
 try {  
 System.*out*.println(path);  
 File dir = path.toFile();  
 if (!dir.exists())  
 dir.mkdir();  
 File file = new File(dir, fileName);  
 if (!file.exists())  
 file.createNewFile();  
 writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(file, true)));  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in creating Cashier file.", e);  
 throw new CashierResourceException("Error's in creating Cashier file.", e);  
 }  
 }  
  
 @PreDestroy  
 private void closeResource() {  
 try {  
 writer.close();  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in closing Cashier file.", e);  
 throw new CashierResourceException("Error's in closing Cashier file.", e);  
 }  
 }  
  
 public void checkout(ShoppingCart cart) {  
 String alert = message.getMessage("alert.inventory.checkout",  
 new Object[]{ cart.getItems(), new Date()}, Locale.*US*);  
 try {  
 writer.write(alert);  
 writer.flush();  
 } catch (IOException e) {  
 //remap to business exception  
 *LOG*.error("Error's in writing on Cashier file.", e);  
 throw new CashierResourceException("Error's in writing on Cashier file.", e);  
 }  
 }  
  
 static class CashierResourceException extends RuntimeException {  
 public CashierResourceException(String message, Exception e) {  
 super(message, e);  
 }  
 }  
  
}

**#@Lazy 로 느긋하게 POJO 초기화하기.**

IoC 컨테이너는 모든 POJO 빈을 조급하기 초기화(Eager Initialization) 한다. 즉, 어플리케이션의 시동과 동시에 POJO 를 초기화한다. @Lazy 를 이용하면 빈을 처음 요청하기 전까지 초기화 과정을 미룬다.

**\*\*\* #@DependsOn 으로 초기화 순서 지정하기**

분산 선언된 POJO 가 서로 참조하다 보면 **경합 조건(race condition)** 이 일어난다. **@DependsOn** 어노테이션을 사용하면 해당 POJO 보다 지정한 POJO 가 먼저 초기화 되도록 순서를 보장한다.

@Bean  
@DependsOn("datePrefixGenerator")  
public SequenceGenerator seqGen() {  
 SequenceGenerator gen = new SequenceGenerator();  
 gen.setPrefix(null);  
 gen.setSuffix("A");  
 gen.setInitial(1000000);  
 return gen;  
}

**\*\*\*\*\* #후처리기(BeanPostProcessor) 로 여러 빈 생성시 초기화, 검증/수정 로직 추가하기.**

빈 후처리기 BeanPostProcessor 를 이용하면 초기화 콜백 메서드 (@Bean의 initeMethod, @PostContruct) 전 후에 원하는 로직을 모든 빈 인스턴스 대상으로 적용할 수 있다. 주로 빈 후처리기는 빈 프러퍼티가 올바른지 체크하거나, 어떤 기준에 따라 빈 프러퍼티를 변경 또는 전체 빈 인스턴스 상대로 어떤 작업을 수행하는 용도로 사용한다. 빈 후처리기는 BeanPostProcessor 인터페이스를 구현한 Bean 컴포넌트이다.

@Component  
public class AuditBeanPostProcessor implements **BeanPostProcessor** {  
  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(AuditBeanPostProcessor.class);  
  
 @Override  
 public Object **postProcessBeforeInitialization**(Object bean, String beanName) throws BeansException {  
 if (bean instanceof Product)  
 *LOG*.debug("IoC container is processing Bean Product [ {} ].", ((Product) bean).getName());  
 **return bean;** }  
  
 @Override  
 public Object **postProcessAfterInitialization**(Object bean, String beanName) throws BeansException {  
 **return bean;** }  
}

**\*\*\* #팩토리로 빈 POJO 생성하기**

* **팩토리 클래스. 런타임시에 특정 객체 생성 로직을 다른 객체로 캡슐화하여 다양성을 넓힌다. 팩토리 클래스의 대표적 예는 DataSource. 클라이언트가 특정 DirverClass 을 런타임시에 주입하면 팩토리 클래스가 알아서 알맞는 Vendor 의 구현체의 인스턴스를 리턴.**

**#정적 팩토리 메서드로 POJO 생성.**

@Component  
@DependsOn("sequenceGenerator")  
public class ProductFactory {  
  
 @Inject  
 private SequenceGenerator seqGen;  
  
 @Value("${special.discount}")  
 private double discount;  
  
 public Product newInstance(String productId) {  
 if ("aaa".equals(productId))  
 return new Battery("AAA", 2.5, seqGen.getSequence());  
 else if ("cdrw".equals(productId))  
 return new Disc("CD-RW", 1.5, seqGen.getSequence(), discount);  
 else if ("dvdrw".equals(productId))  
 return new Disc("DVD-RW", 3.0, seqGen.getSequence(), 3.0);  
 else  
 throw new IllegalArgumentException("Unknown Product");  
 }  
  
}

@Bean  
public Product aaa() {  
 return productFactory.newInstance("aaa");  
}  
  
@Bean  
public Product cdrw() {  
 return productFactory.newInstance("cdrw");  
}  
  
@Bean  
public Product dvdrw() {  
 return productFactory.newInstance("dvdrw");  
}

**#팩토리 빈으로 POJO 생성하기**

**팩토리 빈**은 추상화 클래스 **AbstractFactoryBean<T>** 를 상속하고, **createInstance()** 메서드를 오버라이드해 생성할 빈 인스턴스를 생성한다. 주의할 점은 @Bean 으로 정의한 타입이 AbstractFactoryBean타입이여도 리턴하는 빈 인스턴스는 **getObjectType()** 에 정의된 팩토리가 생성된 빈의 타입이다.

@Configuration  
@PropertySource("classpath:/app/discounts.properties")  
@ComponentScan(basePackages = "com.siwoo.chapter2.shop",  
 includeFilters = @ComponentScan.Filter(  
 type = FilterType.*REGEX*,  
 pattern = "com\\.siwoo.\*bean.\*PostProcessor"  
 ))  
public class ShopConfiguration {  
  
@Value("${special.discount}")  
private double SPECIAL\_DISCOUNT;

@Bean  
 public static PropertySourcesPlaceholderConfigurer  
 propertySourcesPlaceholderConfigurer() {  
 return new PropertySourcesPlaceholderConfigurer();  
 }  
  
 @Bean  
 public ProductFactory productFactory() {  
 ProductFactory factory = new ProductFactory();  
 Map<String, Product> map = new HashMap();  
 map.put("aaa", new Battery("AAA", 2.5, seqService.generate("IT")));  
 map.put("cdrw", new Disc("CD-RW", 1.5, seqService.generate("IT"), SPECIAL\_DISCOUNT));  
 map.put("dvdrw", new Disc("DVD-RW", 3.0, seqService.generate("IT"), -1));  
 factory.setProducts(map);  
 return factory;  
 }  
 //Factory Bean  
 @Bean  
 public DiscountFactoryBean aaaDiscount() {  
 DiscountFactoryBean factoryBean = new DiscountFactoryBean();  
 factoryBean.setProduct(aaa());  
 return factoryBean;  
 }  
 //Factory Bean  
 @Bean  
 public DiscountFactoryBean cdrwDiscount() {  
 DiscountFactoryBean factoryBean = new DiscountFactoryBean();  
 factoryBean.setProduct(cdrw());  
 return factoryBean;  
 }  
 //Factory Bean  
 @Bean  
 public DiscountFactoryBean dvdrwDiscount() {  
 DiscountFactoryBean factoryBean = new DiscountFactoryBean();  
 factoryBean.setProduct(dvdrw());  
 return factoryBean;  
 }  
  
 @Bean  
 public Product aaa() {  
 //return productFactory.newInstance("aaa");  
 return productFactory().getProduct("aaa");  
 }  
  
 @Bean  
 public Product cdrw() {  
 //return productFactory.newInstance("cdrw");  
 return productFactory().getProduct("cdrw");  
 }  
  
 @Bean  
 public Product dvdrw() {  
 //return productFactory.newInstance("dvdrw");  
 return productFactory().getProduct("dvdrw");  
 }  
}

public class DiscountFactoryBean **extends AbstractFactoryBean** {  
  
 private Product product;  
  
 public void setProduct(Product product) {  
 this.product = product;  
 }  
  
 @Override  
 public Class<?> getObjectType() {  
 return Discount.class;  
 }  
  
 @Override  
 protected Object createInstance() throws Exception {  
 Discount discount;  
 if (product instanceof Disc)  
 discount = new Discount(((Disc) product).getDiscount(), true, product.getPrice());  
 else  
 discount = new Discount(0.0D, false, product.getPrice());  
 return discount;  
 }  
  
}

public class Discount {  
  
 private double discount;  
 private boolean adjust;  
 private double price;  
  
 private static final double *PLACES* = Math.*pow*(10, 2);  
  
 public Discount(double discount, boolean adjust, double price) {  
 this.discount = discount;  
 this.adjust = adjust;  
 this.price = price;  
 }  
  
 public double getValue() {  
 if (adjust)  
 return discount > 0 ? Math.*round*((price - (price \* discount)) \* *PLACES*) / *PLACES* : price;  
 else  
 return price;  
 }  
}

@Test  
public void testDiscount() {  
 Discount discount = c.getBean("aaaDiscount", Discount.class);  
 *assertThat*(discount.getValue()).isEqualTo(c.getBean("aaa", Product.class).getPrice());;  
 discount = c.getBean("cdrwDiscount", Discount.class);  
 //discounted  
 *assertThat*(discount.getValue()).isEqualTo(1.35D);  
 discount = c.getBean("dvdrwDiscount", Discount.class);  
 *assertThat*(discount.getValue()).isEqualTo(c.getBean("dvdrw", Product.class).getPrice());  
}

**#다른 환경에서 프로파일(Profile)로 다른 POJO 로드하기**

**POJO 의 초기값은 환경, 시나리오마다 달라질 수 있다. (ex. 개발, 테스트, 운영). 시나리오마다 환경이 다르기 때문에 자바 구성 클래스를 여러개 구성하고 상이한 POJO 를 구성한 다음 자바 구성 클래스에 @Profile 을 설정한 후, 런타임시 해당 시나리오에 맞는 구성 클래스 파일을 읽어들이도록 설정한다.**

@Configuration  
@Profile({"test", "test2"})  
@ComponentScan("com.siwoo.chapter2.shop")  
public class TestShopConfiguration {

**자바 런타임시 매개인자로 프러파일 명시하기.**

**-Dspring.profiles.active=test**

**#POJO에게 IoC 컨테이너 Resource 전달하기.**

빈에서 컨테이너의 리소스를 인지해야 되는 경우, **Aware 인터페이스**를 구현한다.

**Aware 인터페이스 Resource 타입**

**BeanNameAware** 빈 이름

**BeanFactoryAware** 빈 팩토리(BeanFactory).

**ApplicationContextAware** ApplicationContext

**MessageSourceAware** 메시지 소스.

**ApplicationEventPulisherAware** 이벤트 발행시 사용.

**EnvironmentAware** org.springframework.core.env.Environment

public class Cashier implements BeanNameAware {  
  
 @Autowired  
 private MessageSource message;  
  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(Cashier.class);  
  
 private Writer writer;  
 @Value("classpath:/app")  
 private Path path;  
  
 @Value("checkout")  
 private String fileName;  
  
 public Cashier() {  
 }  
  
 public Cashier(Path path, String fileName) {  
 this.path = path;  
 this.fileName = fileName;  
 }

//BeaNameAware 메서드 구현  
 @Override  
 public void setBeanName(String name) {  
 this.fileName = name;  
 }

**#어노테이션을 활용한 AOP 프로그래밍**

* **AOP –** 기능을 **핵심** **비즈니스 로직(business logic)**과 **공통 모듈(common module:보안, Logging, Transaction)** 로 구분하고, 핵심 로직에 영향을 미치지 않고 사이사이에 공통 모듈을 효과적으로 잘 끼워놓도록 하는 개발 방법.

**Aspect (Advice + Pointcut)** 을 정의하려면 자바 클래스에 **@Aspect** 을 붙이고 메서드별로 적절한 어노테이션을 붙여 **어드바이스(Advice)** 를 만든다. 어드바이스 어노테이션의 종류는 **@Before, @After, @AfterReturning, @AfterThrowing, @Around** 등이 있다.

**IoC 컨테이너**에서 **어노테이션을 통한Aspect 기능**을 활성화 하려면 **@EnableAspectJAutoProxy** 를 붙인다**. @EnableAspectJAutoProxy** 에서 **proxyTargetClass** 속성을 true로 설정하면 동적 프록시 (JDK) 대신 **CGLIB**를 사용한다.

#기본적으로 스프링은 인터페이스 기반의 **JDK 동적 프록시를** 사용한다.

#인터페이스를 사용할 수 없는 경우라면 **CGLIB 프록시 기능을** 사용해야 한다.

**#Aspect, Advice, PointCut?**

**Aspect(Advice + PointCut**) 은 여러 객체의 **공통 관심사(로깅, 트랜잭션) 을 모듈화**한 자바 클래스로, **@Aspect**을 붙여 표시한다.

**Advice 는 @Before, @After, @AfterReturning, @AfterThrowing, @Around** 를 붙인 자바 메서드로, 무엇을 할 것인지를 정의한다.

**PointCut** 은 어드바이스에 적용할 타입 및 객체를 찾는 표현식이다.

**#@Before 어드바이스**

**Before Advice**는 특정 프로그램 **실행 지점 이전**의 공통 관심사를 처리하는 메서드로, **@Before** 를 붙이고 **포이트컷 표현식(PointCut Expression)** 을 어노테이션의 값으로 지정한다.

* **PointCut 표현식** **execution 안의 AspectJ 표현식.**

**와일드카드(\*)** 는 모든 값를 매치한다.

**..** 은 0개 이상의 모든 값을 매치한다.

// Aspect 공통 모듈임을 선언한다. (빈 스캐닝으로 빈 선언시 @Component 어노테이션 필요)

@Aspect  
public class LoggingCalculationAspect {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(LoggingCalculationAspect.class);

//어드바이스 선언과 동시에 PointCut 으로 적용 지점 지정  
 @Before("execution(\* \*..ArithmeticCalculator\*.add(..))")  
 public void logBefore() {  
 *log*.info("Logging Before add() method.");  
 }  
  
}

@Configuration

//Aspect Annotation 기능 활성화  
@EnableAspectJAutoProxy  
@ComponentScan(basePackages = "com.siwoo.springboot.calculator")  
public class CalculatorConfiguration {  
  
 @Bean  
 public LoggingCalculationAspect loggingCalculationAspect() {  
 return new LoggingCalculationAspect();  
 }  
  
}

**#조인포인트(JoinPoint)?**

포인트컷으로 매칭한 **각 실행 지점을 조인포인트(JoinPoint )** 라고 한다. **포이트컷(PointCut)**은 여러 조인포인트를 매치하기 위한 표현식이다. 어드바이스에서 특정 **JoinPoint 의 세부 사항에** 액세스하려면 **JoinPoint 형의 인수를** 어드바이스 메서드에서 인자로 전달받아야 한다.

@Aspect  
public class LoggingCalculationAspect {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(LoggingCalculationAspect.class);  
  
 **@Before("execution(\* \*..calculator..\*(..))")**

**//특정 실행 지점의 세부사항 (JoinPoint)**  
 **public void logBefore(JoinPoint joinPoint) {** *log*.info("The method " + joinPoint.getSignature().getName() + "() begins with args " + Arrays.*toString*(joinPoint.getArgs()) + ".");  
 }  
  
}

**#After 어드바이스 와 AfterReturning**

**After 어드바이스**는 **JoinPoint** 가 끝나면 실행되는 메서드로, **@After**를 붙여 지정한다.

조인포인트가 정상 실행되든, 도중에 예외가 발생하든 상관없이 실행된다.

**AfterReturning 어드바이스는** 조인포인트가 값을 반환한 경우(정상 종료) 한 경우에만 작동한다. 리턴한 결과값을 가져오려면 **@AfterReturning 어노테이션의 returning 속성**으로 변수명을 인수로 지정한다.

//타겟 메서드의 성공여부와 상관없이 실행

@After("execution(\* \*..calculator..\*(..))")  
public void logAfter(JoinPoint joinPoint) {  
 *log*.info("The method {}() ends.", joinPoint.getSignature().getName());  
}  
 //타겟 메서드가 정상 종류된 이후에만 실행.   
@AfterReturning(value = "execution(\* \*..calculator..\*(..))", returning = "result")  
public void logAfterReturning(JoinPoint joinPoint, Object result) {  
 *log*.info("The method {}() ends, result {}", joinPoint, result);  
}

**#AfterThrowing 어드바이스.**

**After Throwing 어드바이스**는 조인포인트 실행 도중 예외가 날 경우에만 실행된다.발생한 예외는 **@AfterThrowing 속성인 throwing**에 변수명을 지정하여 전달받을 수 있다. 특정 예외가 발생시만 어드바이스를 실행할 시, 주어진 예외 타입과 호환되는 타입을 인수로 지정한다.

@AfterThrowing(value = "execution(\* \*..calculator..\*(..))", throwing = "e")  
public void logException(JoinPoint joinPoint, Throwable e) {  
 *log*.info("An exception {} has been thrown in {}", e, joinPoint.getSignature().getName());  
}  
  
@AfterThrowing(value = "execution(\* \*..calculator..\*(..))", throwing = "e")

// IllegalArgumentException 예외 발생시에만 작동.  
public void logException(JoinPoint joinPoint, IllegalArgumentException e) {  
 *log*.info("An IllegalArgumentException {} has been thrown in {}", e, joinPoint.getSignature().getName());  
}

**#Around 어드바이스.**

**Around 어드바이스**는 조인포인트의 모든 시점을 컨트롤한다. 또한 원본 조인포인트를 언제 실행할지, 실행 자체를 할지 말지, 원본 인수값, 최종 반환값 까지 변경 가능하다.

* 원본 조인포인트를 진행하는 호출 **ProceedingJoinPoint.proceed()** 을 항상 기억하자\

@Aspect  
public class LoggingCalculationAspect {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(LoggingCalculationAspect.class);  
  
 **@Around("execution(\* \*..calculator..\*(..))")** public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {  
 *log*.info("The method {}() begins with args {}", joinPoint.getSignature().getName(), Arrays.*toString*(joinPoint.getArgs()));  
 try {

// 타겟 메서드 실행 지점. **Object result = joinPoint.proceed();**  
 *log*.info("The method {}() ends with result {}", joinPoint.getSignature().getName(), result);  
 return result;  
 } catch (Throwable e) {  
 *log*.info("The exception {}() has bean thrown in {}", e, joinPoint.getSignature().getName());  
 throw e;  
 }  
  
 }

}

**#조인 포인트의 정보 가져오기. (JoinPoint’s Detail)**

Aspect 는 내부적으로 Proxy (타겟을 감싼 객체) 을 사용한다. JoinPoint.getThis() 는 Proxy을 리턴하며 JoinPoint.getTarget() 실제 타겟 객체를 리턴한다.

@Before("execution(\* \*..calculator..\*(..))")  
public void exploreJoinPoint(JoinPoint joinPoint) {  
 *log*.info("Advice 종류: {}", joinPoint.getKind());  
 *log*.info("타겟 클래스: {}", joinPoint.getSignature().getDeclaringType());  
 *log*.info("메서드 Signature: {}", joinPoint.getSignature());  
 *log*.info("메서드 인자: {}", Arrays.*toString*(joinPoint.getArgs()));

// Proxy's 타겟: com.siwoo.springboot.calculator.ArithmeticCalculatorImpl  
 *log*.info("Proxy's 타겟: {}", joinPoint.getTarget().getClass().getName());

// Proxy: com.sun.proxy.$Proxy22  
 *log*.info("Proxy: {}", joinPoint.getThis().getClass().getName());  
}

**#@Order 로 Aspect 순서 지정하기.**

**공통 모듈(Aspect)** 이 여러개일 경우, 어느 쪽을 먼저 적용할지를 **@Order 어노테이션**을 통해 지정할 수 있다. 이 경우 **숫자가 적을수록 우선순위가 높다**.

@Aspect **@Order(0)**  
public class ValidateCalculationAspect {  
  
 @Before("execution(\* \*..calculator..\*(double, double))")  
 public void validateBefore(JoinPoint joinPoint) {  
 for (Object arg: joinPoint.getArgs()) {  
 validate((Double) arg);  
 }  
 }  
  
 private void validate(Double value) {  
 if (value.isNaN() || value.isInfinite() || value < 0)  
 throw new IllegalArgumentException("Positive numbers only");  
 }  
}

@Aspect **@Order(1)**  
public class LoggingCalculationAspect {

…

}

**#Aspect 포인트컷 재사용하기.**

하나의 **포인트컷 표현식을** 여러 **Advice** 에서 사용하는 경우, **@Pointcut** 으로 포인트컷을 따로 정의하면 **재사용**할 수 있다. 포인트컷을 참조할 때는 클래스명을 지정하며, 현재 Aspect 와 다른 패키지에 있다면 패키지명 혹은 **fully qualified name** 을 기재한다.

public class Pointcuts {  
  **// 포인트컷 정의  
 @Pointcut("execution(\* \*..calculator..\*(..))")  
 public void calculatorPointcut() {}**

**@Pointcut("execution(\* \*..calculator..\*(double, double))")  
 public void calculatorWithDoubleArgsPointcut() {}**  
  
}

@Aspect  
@Order(0)  
public class ValidateCalculationAspect {  
 // 정의된 포인트컷 지정.  
 @Before("Pointcuts.calculatorWithDoubleArgsPointcut()")  
 public void validateBefore(JoinPoint joinPoint) {  
 for (Object arg: joinPoint.getArgs()) {  
 validate((Double) arg);  
 }  
 }

@Aspect  
@Order(1)  
public class LoggingCalculationAspect {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(LoggingCalculationAspect.class);  
  
 @Before("Pointcuts.calculatorPointcut()")  
 public void exploreJoinPoint(JoinPoint joinPoint) {  
 *log*.info("Advice 종류: {}", joinPoint.getKind());  
 *log*.info("타겟 클래스: {}", joinPoint.getSignature().getDeclaringType());  
 *log*.info("메서드 Signature: {}", joinPoint.getSignature());  
 *log*.info("메서드 인자: {}", Arrays.*toString*(joinPoint.getArgs()));  
 *log*.info("Proxy's 타겟: {}", joinPoint.getTarget().getClass().getName());  
 *log*.info("Proxy: {}", joinPoint.getThis().getClass().getName());  
 }  
  
 @Around("Pointcuts.calculatorPointcut()")  
 public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {  
 *log*.info("The method {}() begins with args {}", joinPoint.getSignature().getName(), Arrays.*toString*(joinPoint.getArgs()));  
 try {  
 Object result = joinPoint.proceed();  
 *log*.info("The method {}() ends with result {}", joinPoint.getSignature().getName(), result);  
 return result;  
 } catch (Throwable e) {  
 *log*.info("The exception {}() has bean thrown in {}", e, joinPoint.getSignature().getName());  
 throw e;  
 }  
 }

**#AspectJ 포인트컷 표현식**

**공통 관심사는** 프로그램이 실행되는 지점, 즉 여러 **조인포인트**에 걸쳐 분포한다**. AspectJ**는 다양한 종류의 **조인포인트를 매치할 수 있는 표현식 언어를** 제공한다. **스프링 AOP**에서는 **AspectJ 포인트컷 표현식을** 활용해 포인트컷을 정의하며 **runtime** 에 **AspectJ 라이브러리**를 이용해 포인트컷 표현식을 해석한다.

* AspectJ 포인트컷 표현식은 IoC 컨테이너 안의 빈에만 조인포인트를 지원.

**#메서드 시그니처 패턴(Method Signature Pattern)**

**execution(\* com.siwoo.springboot.calculator.ArithmeticCalculator.\*(..))**

앞 쪽의 \* 은 모든 **Access Modifier(public, protected, private) 와 반환형**을 매치 시킨다.

**()** 괄호안의 **..** 은 모든 인자를 매치시킨다.

**execution(\* ArithmeticCalculator.\*(..))**

ArithmeticCalculator 인터페이스에 선언된 모든 public 메서드를 매치시킨다. 대상 클래스와 Aspect가 같은 패키지가 있다면 패키지 매치는 생략가능.

**execution(public \* ArithmeticCalculator.\*(..))**

**Access Modifier** 가 public 인 메서드만 매치시킨다.

**execution(public double ArithmeticCalculator.\*(..))**

**Access Modifier** 가 public 이고 반환값이 double 인 메서드만 매치시킨다.

**execution(public double ArithmetricCalculator.\*(double, ..))**

첫 번째가 인수가 double 인 메서드를 매치시킨다. 이후의 인수는 모두 매치된다.

**#커스텀 어노테이션을 이용한 조인포인트 매치.**

커스텀 메서드/타입 레벨에 커스텀 어노테이션을 붙혀 해당 조인포인트를 매치시킬 수 있다.

* 어노테이션은 상속되지 않으므로, 인터페이스가 아닌 구현 클레스에 붙여야 한다.
* 클래스 레벨에 붙이면 모든 메서드에 적용된다.

**@Target({ElementType.*METHOD*, ElementType.*TYPE*})  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
public @interface Logging {  
  
}**

@Component("arithmetricCalculator")  
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {  
 **@Logging**  
 @Override  
 public double add(double a, double b) {  
 double result = a + b;  
 System.*out*.println("a" + " + " + b + " = " + result);  
 return result;  
 }

}

**@Aspect  
@Order(1)**  
public class LoggingCalculationAspect {  
  
 private static final Logger *log* = LoggerFactory.*getLogger*(LoggingCalculationAspect.class);  
 //annotation 안에 포인트컷 표현식 지정.  
 **@Before("@annotation(com.siwoo.springboot.aop.Logging)")**  
 public void exploreJoinPoint(JoinPoint joinPoint) {  
 *log*.info("Advice 종류: {}", joinPoint.getKind());  
 *log*.info("타겟 클래스: {}", joinPoint.getSignature().getDeclaringType());  
 *log*.info("메서드 Signature: {}", joinPoint.getSignature());  
 *log*.info("메서드 인자: {}", Arrays.*toString*(joinPoint.getArgs()));  
 *log*.info("Proxy's 타겟: {}", joinPoint.getTarget().getClass().getName());  
 *log*.info("Proxy: {}", joinPoint.getThis().getClass().getName());  
 }

**#타입 시그니처 패턴.**

타입 시그니처 패턴은 특정한 타입 내부의 모든 조인포인트를 매치한다. within() 을 적용하면 그 타입 안에 구현된 메서드를 실행할 때만 어드바이스를 적용한다.

**within(com.siwoo.springboot.calculator.\*)**

**해당 패키지 안의 모든 메서드에 실행 조인포인트를 매치한다.**

**within(com.siwoo.springboot.calculator..\*)**

**해당 패키지와 하위 패키지의 모든 메서드에 실행 조인포인트를 매치한다.**

**within(com.siwoo.springboot.calculator.ArithmetricCalculator)**

**특정 한 클래스 내부에 구현된 메서드 실행 조인포인트를 매치한다.**

**within(ArithmeticCalculator+)**

**해당 인터페이스를 구현한 모든 클래스의 메서드 실행 조인포인트를 매치한다.**

//@Before("within(com.siwoo.springboot.calculator.\*)")  
//@Before("within(com.siwoo.springboot.calculator..\*)")  
//@Before("within(com.siwoo.springboot.calculator.ArithmeticCalculatorImpl)")  
@Before("within(com.siwoo.springboot.calculator.ArithmeticCalculator+)")  
public void exploreJoinPoint(JoinPoint joinPoint) {  
 *log*.info("Advice 종류: {}", joinPoint.getKind());  
 *log*.info("타겟 클래스: {}", joinPoint.getSignature().getDeclaringType());  
 *log*.info("메서드 Signature: {}", joinPoint.getSignature());  
 *log*.info("메서드 인자: {}", Arrays.*toString*(joinPoint.getArgs()));  
 *log*.info("Proxy's 타겟: {}", joinPoint.getTarget().getClass().getName());  
 *log*.info("Proxy: {}", joinPoint.getThis().getClass().getName());  
}

**#포인트컷 표현식 조합하기.**

AspectJ포인트컷 표현식은 **&&(and), ||(or), !(not)** 등의 연산자로 조합할 수 있다**.**

**//ArithmetricCalulator 혹은 UnitCalulator 을 구현한 모든 클래스의 메서드에 매치시킨다.**

**@Before("within(com.siwoo.springboot.calculator.ArithmeticCalculator+) || within(com.siwoo.springboot.calculator.UnitCalculator+)")**  
public void exploreJoinPoint(JoinPoint joinPoint) {  
 *log*.info("Advice 종류: {}", joinPoint.getKind());  
 *log*.info("타겟 클래스: {}", joinPoint.getSignature().getDeclaringType());  
 *log*.info("메서드 Signature: {}", joinPoint.getSignature());  
 *log*.info("메서드 인자: {}", Arrays.*toString*(joinPoint.getArgs()));  
 *log*.info("Proxy's 타겟: {}", joinPoint.getTarget().getClass().getName());  
 *log*.info("Proxy: {}", joinPoint.getThis().getClass().getName());  
}

**#Introduction 을 이용한 POJO에 기능 더하기**

**Introduction 은 AOP Advice 의 특별한 타입이다**. 객체가 어떤 인터페이스의 구현 클래스를 런타임에 공급받아 동적으로 특정 인터페이스를 구현하는 기술이다. 마치 객체가 런타임에 구현 클래스를 상속하는 것처럼 보인다. (다중 상속 가능한 것 처럼 보인다.)

* **Introuction 의 사용 예**

예를 들어, UsageTracked 인터페이스와 DefaultUsageTracked 인터페이스를 구현하는 경우, 다음 측면에서는 서비스 인터페이스의 모든 구현자도 UsageTracked 인터페이스를 구현한다고 선언합니다. (예를 들어 JMX를 통해 통계를 노출하려면)

@Aspect  
public class EnhanceCalculationIntroduction {

// ArithmeticCalculatorImpl 서비스를 동적으로 MaxCalculator 인터페이스를 구현하도록 하며,

// 구현 클래스는 MaxCalculatorImpl 로 설정  
 **@DeclareParents(value = "com.siwoo.springboot.calculator.ArithmeticCalculatorImpl",  
 defaultImpl = MaxCalculatorImpl.class)  
 public MaxCalculator maxCalculator;  
  
 @DeclareParents(value = "com.siwoo.springboot.calculator.ArithmeticCalculatorImpl",  
 defaultImpl = MinCalculatorImpl.class)  
 public MinCalculator minCalculator;**  
  
}

@Component("arithmetricCalculator")  
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

}

@Test  
public void testBeforeAdvice() {  
 **ArithmeticCalculator arithCal = c.getBean(ArithmeticCalculator.class);**

**// 동적으로 구현한 인터페이스로 캐스팅.  
 MaxCalculator maxCalculator = (MaxCalculator) arithCal;**  
 maxCalculator.max(10, 5);  
 MinCalculator minCalculator = (MinCalculator) arithCal;  
 minCalculator.min(10, 5);  
  
}