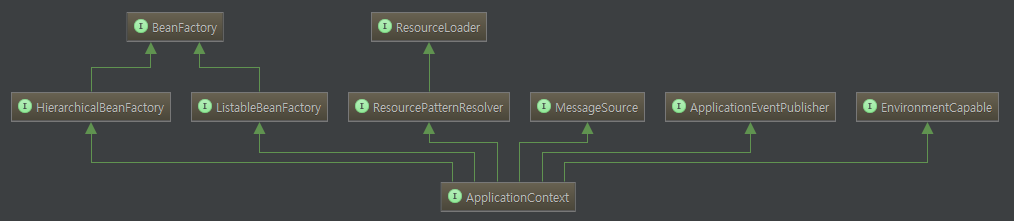
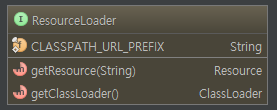
**ApplicationContext 전체 구조**

****

**1) ResourceLoader**

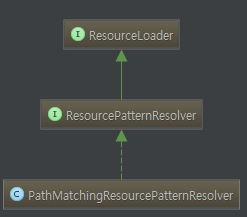
****

Resource 인스턴스를 로딩할 수 있는 객체

**Resource 종류**

|  |  |
| --- | --- |
| **Resource 명** | **설 명** |
| UrlResource | URL로 얻어와야 하는 리소스 (java.net.URL) |
| ClassPathResource | 클래스패스(resource)에서 얻어와야 하는 리소스 |
| FileSystemResource | file:/ (java.io.File) |
| ServletContextResource | 웹 어플리케이션의 루트 경로 |
| InputStreamResource | InputStream에 대한 리소스 (파일, 버퍼, 데이터) |
| ByteArrayResource | 바이트 배열에 대한 리소스 |

**2) ResourcePatternResolver extends ResourceLoader**

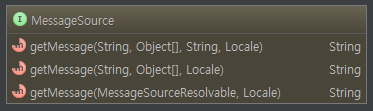


**얜 뭐임?**

앤트스타일로 읽어 오는 애임 ㅇㅇ

ex) "/WEB-INF/\*-context.xml")

**3) MessageSource**



다국어 처리 할 때 사용한다. (로케일 사용)

ex)

**message\_ko\_kr.properties**

안녕하세요.

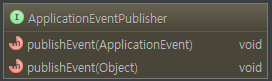
**message\_en\_us.properties**

Hello

**message\_ja\_jp.properties**

こんにちわ

**4) ApplicationEventPublisher**



이벤트 지원

|  |  |
| --- | --- |
| **이벤트 종류** | **용도** |
| ApplicationEventPublisher | 이벤트 발신자 |
| ApplicationListener | 이벤트 수신자 |
| ApplicationEvent | 이벤트 |

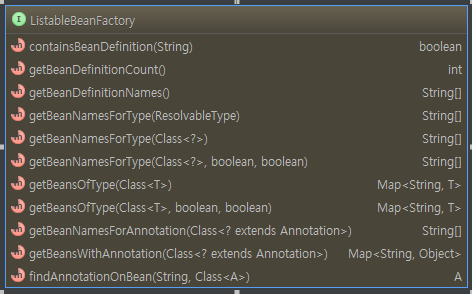
**5) EnvironmentCapable**



Profile에 대한 선택, property에 대한 통합접근 제공

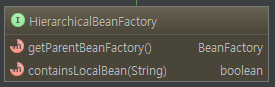
GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();  
ctx.getEnvironment().setActiveProfiles("dev");  
ctx.load("classpath:/\*-config.xml");  
ctx.refresh();

**6) ListableBeanFactory (열거하다) extends BeanFactory**



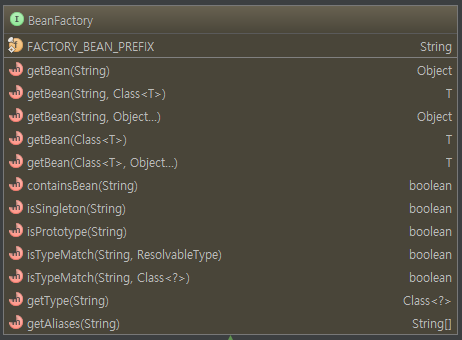
BeanFactory의 서브인터페이스 (기능 확장)

**7) HierarchicalBeanFactory (계층적) extends BeanFactory**



BeanFactory의 서브인터페이스 (기능 확장)

**8) BeanFactory (lazyloading)**



Bean의 생성과 소멸 담당

Bean 생성 시 필요한 속성 설정

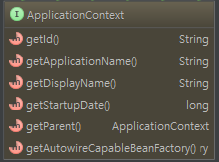
Bean의 LifeCycle에 관련된 메소드 호출

다수의 BeanFactory 인터페이스 구현 클래스를 제공

BeanFactory factory = new XmlBeanFactory(new FileInputStream("bean.xml"));  
MyBean myBean = (Mybean) factory.getBean("myBean");

getBean이 호출되면, 팩토리는 의존성 주입을 이용해 빈을 인스턴스화 한다.

**9) ApplicationContext (preloading)**



BeanFactory의 모든 기능 제공

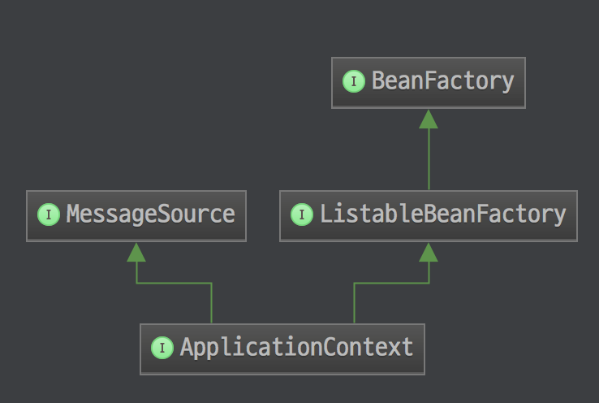
ResourceBundle 파일을 이용한 국제화(I18N) 지원

다양한 Resource 로딩 방법 제공

이벤트 핸들링

**스프링의 히스토리**

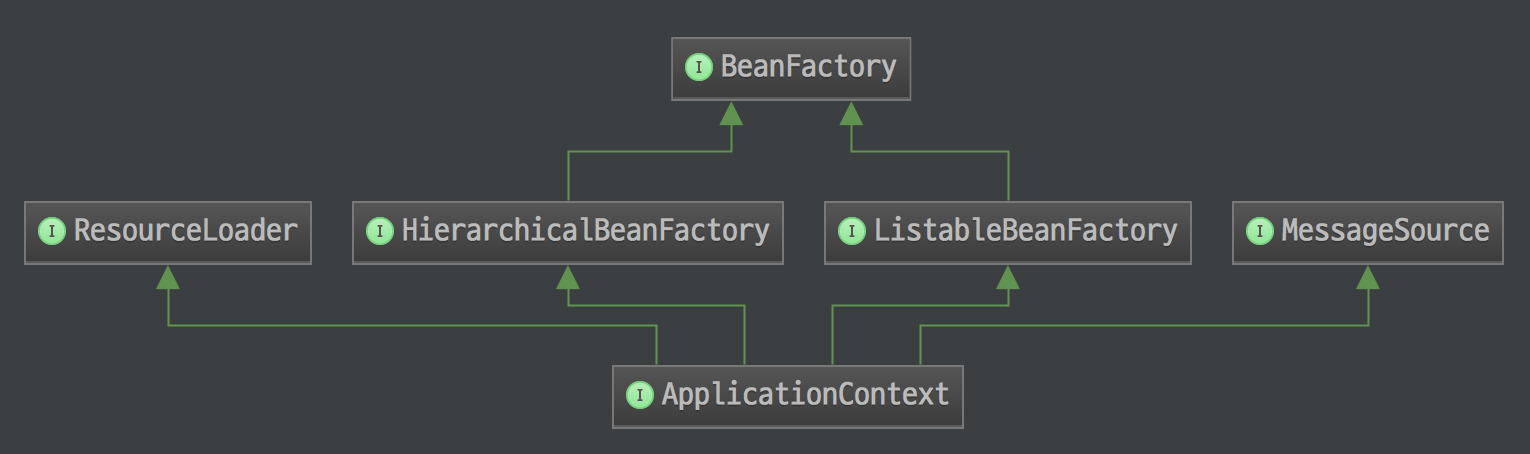
**ver 0.9**



BeanFactory의 Sub Interface인 **HierarchicalBeanFactory** 가 없었고, ApplicationContext의

초기 기능은 **MessageSource**(국제화 뿐이였다.)

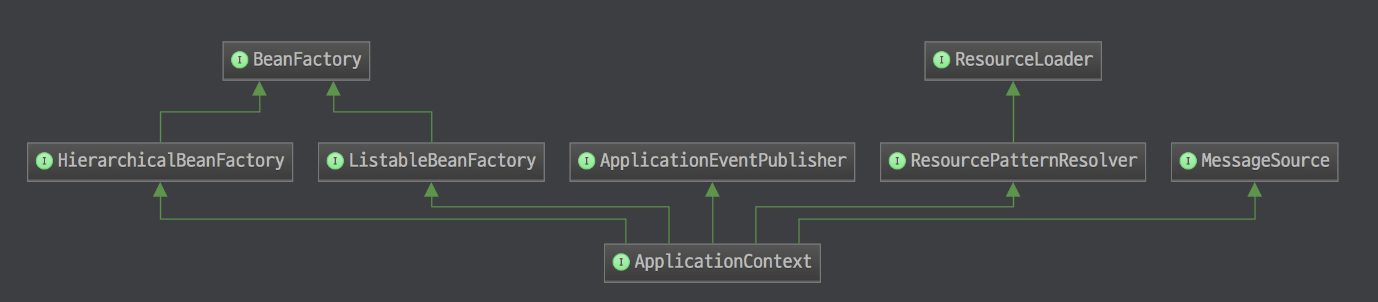
**ver 1.0**



BeanFactory의 Sub Interface인 **HierarchicalBeanFactory**가 추가되었고,

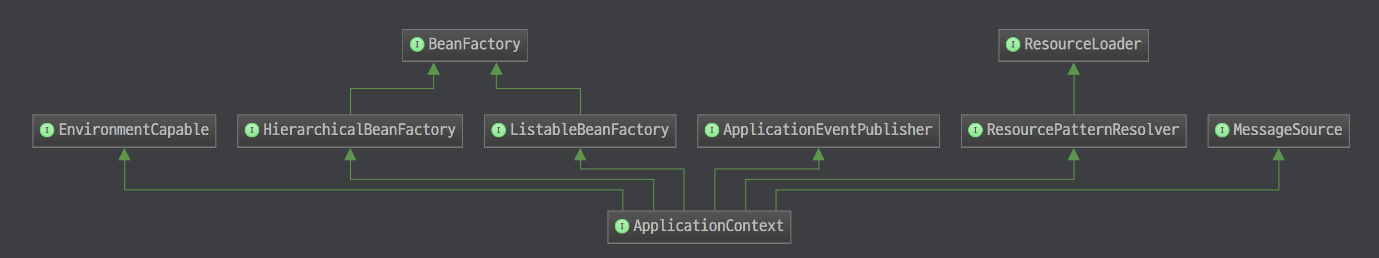
**ResourceLoader**가 추가되었다.

**ver 2.0**



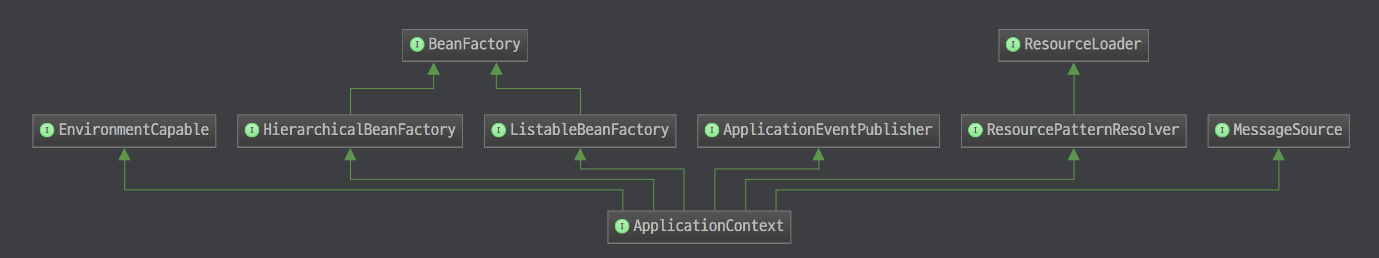
**ResourcePatternResolver**(앤트처럼 리소스 불러오는애) 가 추가되었고, **ApplicationEventPublisher**가 추가되었다.

**ver 3.0**



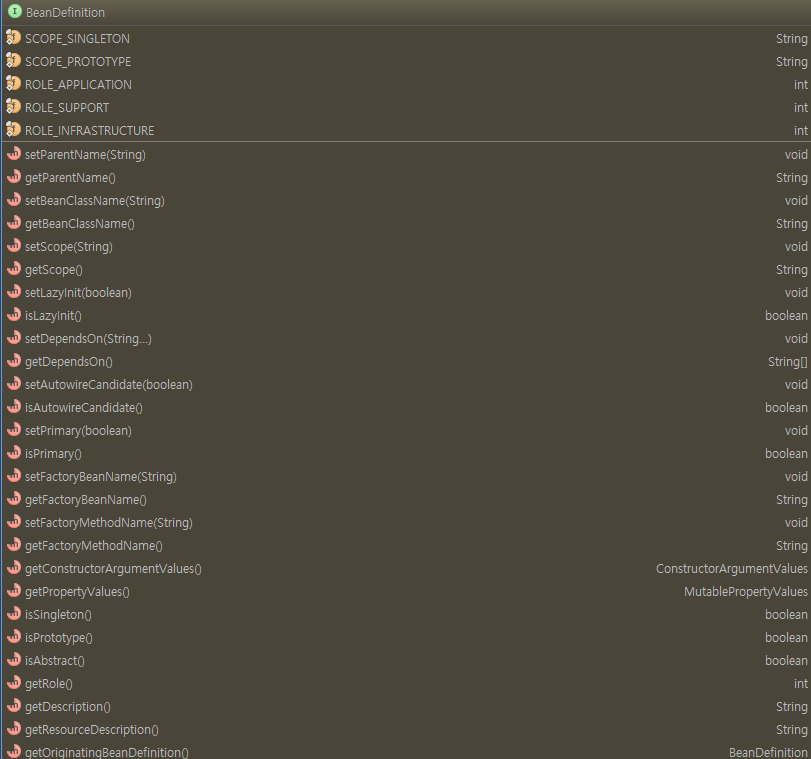
**EnvironmentCapable**이 추가되었다.

**ver 4.0 (동)**



**BeanDefinition과 BeanDefinitionReader**

**BeanDefinition**

****



**BeanDefinition**

Spring의 빈 설정은 XML과 JavaConfig(Annotation)으로 구성되어 잇다.

이 말은 사실 BeanDefinition으로 추상화되어 있는 빈설정을 xml, JavaConfig로 표현하고 있다고 말할 수 있다.

**BeanFactory vs ApplicationContext 심화**

**1. Bean 등록은 어떻게 될까?**

**1) AnnotationConfigApplicationContext 통해 Bean등록**

public class ApplicationContextMain {  
 public static void main(String[] args) {  
 ApplicationContext applicationContext =

new AnnotationConfigApplicationContext(BeanConfig.class);  
 }  
}

**2) AnnotationConfigApplicationContext 생성자 (register, refresh)**

public class AnnotationConfigApplicationContext extends GenericApplicationContext implements AnnotationConfigRegistry {  
 public AnnotationConfigApplicationContext(Class... annotatedClasses) {  
 this();  
 this.register(annotatedClasses);  
 this.refresh();  
 }

**3) 타고 가다 보면 registerBean 출현 (가장 중요)**

public void registerBean(Class<?> annotatedClass, String name, Class... qualifiers) {  
 AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);  
 if(!this.conditionEvaluator.shouldSkip(abd.getMetadata())) {  
 ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(abd);  
 abd.setScope(scopeMetadata.getScopeName());  
 String beanName = name != null?name:this.beanNameGenerator.generateBeanName(abd, this.registry);  
 AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);  
 if(qualifiers != null) {  
 Class[] definitionHolder = qualifiers;  
 int var8 = qualifiers.length;  
  
 for(int var9 = 0; var9 < var8; ++var9) {  
 Class qualifier = definitionHolder[var9];  
 if(Primary.class == qualifier) {  
 abd.setPrimary(true);  
 } else if(Lazy.class == qualifier) {  
 abd.setLazyInit(true);  
 } else {  
 abd.addQualifier(new AutowireCandidateQualifier(qualifier));  
 }  
 }  
 }  
  
 BeanDefinitionHolder var11 = new BeanDefinitionHolder(abd, beanName);  
 var11 = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, var11, this.registry);  
 BeanDefinitionReaderUtils.registerBeanDefinition(var11, this.registry);  
 }  
}

**-AnnotatedGenericBeanDefinition :** BeanDefinitaion 구현체중 하나, BeanConfig를 통해 등록될 Bean에 메타데이터를 정의

public void registerBean(Class<?> annotatedClass, String name, Class... qualifiers) {  
 AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);

...

}

**-ScopeMetadata** : Bean Scope에 메타데이터를 정의 (기본 singleton)

public class ScopeMetadata {  
 private String scopeName = "singleton";

. . .  
}

**- AnnotationBeanNameGenerator :** bean name을 가지고온다.

String beanName = name != null?name:this.beanNameGenerator.generateBeanName(abd, this.registry);

**AnnotationBeanNameGenerator.class (Bean 이름 있으면 쓰고 없으면 class 이름으로)**

public String generateBeanName(BeanDefinition definition, BeanDefinitionRegistry registry) {  
 if(definition instanceof AnnotatedBeanDefinition) {  
 String beanName = this.determineBeanNameFromAnnotation((AnnotatedBeanDefinition)definition);  
 if(StringUtils.hasText(beanName)) {  
 return beanName;  
 }  
 }  
  
 return this.buildDefaultBeanName(definition, registry);  
}

...

protected String buildDefaultBeanName(BeanDefinition definition) {  
 String shortClassName = ClassUtils.getShortName(definition.getBeanClassName());  
 return Introspector.decapitalize(shortClassName);  
}

**-AnnotationConfigUtils :** lazy, Primary, DependsOn 등 어노테이션 체크해서 설정

AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);

**-Qualifires도 체크함**

if(qualifiers != null) {  
 Class[] definitionHolder = qualifiers;  
 int var8 = qualifiers.length;  
  
 for(int var9 = 0; var9 < var8; ++var9) {  
 Class qualifier = definitionHolder[var9];  
 if(Primary.class == qualifier) {  
 abd.setPrimary(true);  
 } else if(Lazy.class == qualifier) {  
 abd.setLazyInit(true);  
 } else {  
 abd.addQualifier(new AutowireCandidateQualifier(qualifier));  
 }  
 }  
}

**-** **BeanDefinitionHolder :** beandefinition, beanName, aliases 등 정의

BeanDefinitionHolder var11 = new BeanDefinitionHolder(abd, beanName);

**-AnnotationConfigUtils.applyScopedProxyMdoe :** 프록시 생성 여부

AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, var11, this.registry);

1) NO : 프록시 사용 안함

2) JDK 다이나믹 프록시(INTERFACES) : Spring에서 프록시 대상의 객체가 최소 하나의 인터페이스를 구현했을 때

3) Cglib(TARGET\_CLASS) : 인터페이스를 구현하지 않았을 때

**-BeanDefinitionReaderUtils.registerBeanDefinition :** Bean을 Spring Container에 저장하는 역할을 하는 utils 클래스

BeanDefinitionReaderUtils.registerBeanDefinition(var11, this.registry);

**-** **GenericApplicationContext에서 DefaultListableBeanFactory 호출**

public class GenericApplicationContext extends AbstractApplicationContext implements BeanDefinitionRegistry {  
 private final DefaultListableBeanFactory beanFactory;  
  
 public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition) throws BeanDefinitionStoreException {  
 this.beanFactory.registerBeanDefinition(beanName, beanDefinition);  
 }  
}

🡺ApplicationContext(GenericApplicationContext)는

BeanFactory(DefaultListableBeanFacotry)를 상속과 동시에 위임해서 사용하고 있다.

**-DefaultListableBeanFactory.registerBeanDefinition :** 실제로 Bean을 beanDefinitionMap에 저장하는 부분

public class DefaultListableBeanFactory ... {  
  
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap(256);  
  
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)  
{  
 ...  
 this.beanDefinitionMap.put(beanName, beanDefinition);  
 this.beanDefinitionNames.add(beanName);  
 ...  
 }  
}

🡺드디어 빈을 등록했다. 한가지 이상한게 있다. 왜 Bean instance가 아닌 BeanDefinition을

저장하는 것일까?

Spring Container는 BeanDefinition을 저장하고 있다가, getBean 메소드가 호출 될 때

Scope를 판단해서 instance를 생성 🡺 저장(singleton 인경우만) 🡺 반환 한다.

singelton이 아닌 경우를 생각하면 BeanDefinition을 저장하고 있다가 새로운 객체를 생성해서 반환해야 되기 때문이다.

🡺빈 등록은 ApplicationContext에서 직접 하는 것이 아니라, BeanFactory의 구현체를 이용해서 위임했다.

**2. 그러면 Bean을 어떻게 반환할까?**

그러면 앞에서 등록한 BeanDefinition을 어떻게 instance화 후 반환 하는 것일까?

**getBean을 이용해서 Bean을 불러 오자**

public class ApplicationContextMain {  
 public static void main(String[] args) {  
 ApplicationContext applicationContext = new AnnotationConfigApplicationContext(BeanConfig.class);  
 Entitlement entitlement = applicationContext.getBean(Entitlement.class);  
 }  
}

AnnotationConfigApplicationContext 의 getBean은 부모 클래스인 AbstractApplicationContext에 구현되어 있다.

**AbstractApplicationContext의 getBean**

public abstract class AbstractApplicationContext {

...

public <T> T getBean(Class<T> requiredType) throws BeansException {  
 this.assertBeanFactoryActive();  
 return this.getBeanFactory().getBean(requiredType);  
 }

}

**GenericApplicationContext.class**

public class GenericApplicationContext {

...

public final ConfigurableListableBeanFactory getBeanFactory() {  
 return this.beanFactory;  
 }

}

GenericApplicationContext는 **BeanFactory**와 **ApplicationContext**에 구현체인 동시에

DefaultListableBeanFactory 를 멤버변수로 선언하며 구현체로 사용하고 있다.

public class GenericApplicationContext {

...

public final DefaultListableBeanFactory getDefaultListableBeanFactory() {  
 return this.beanFactory;  
 }

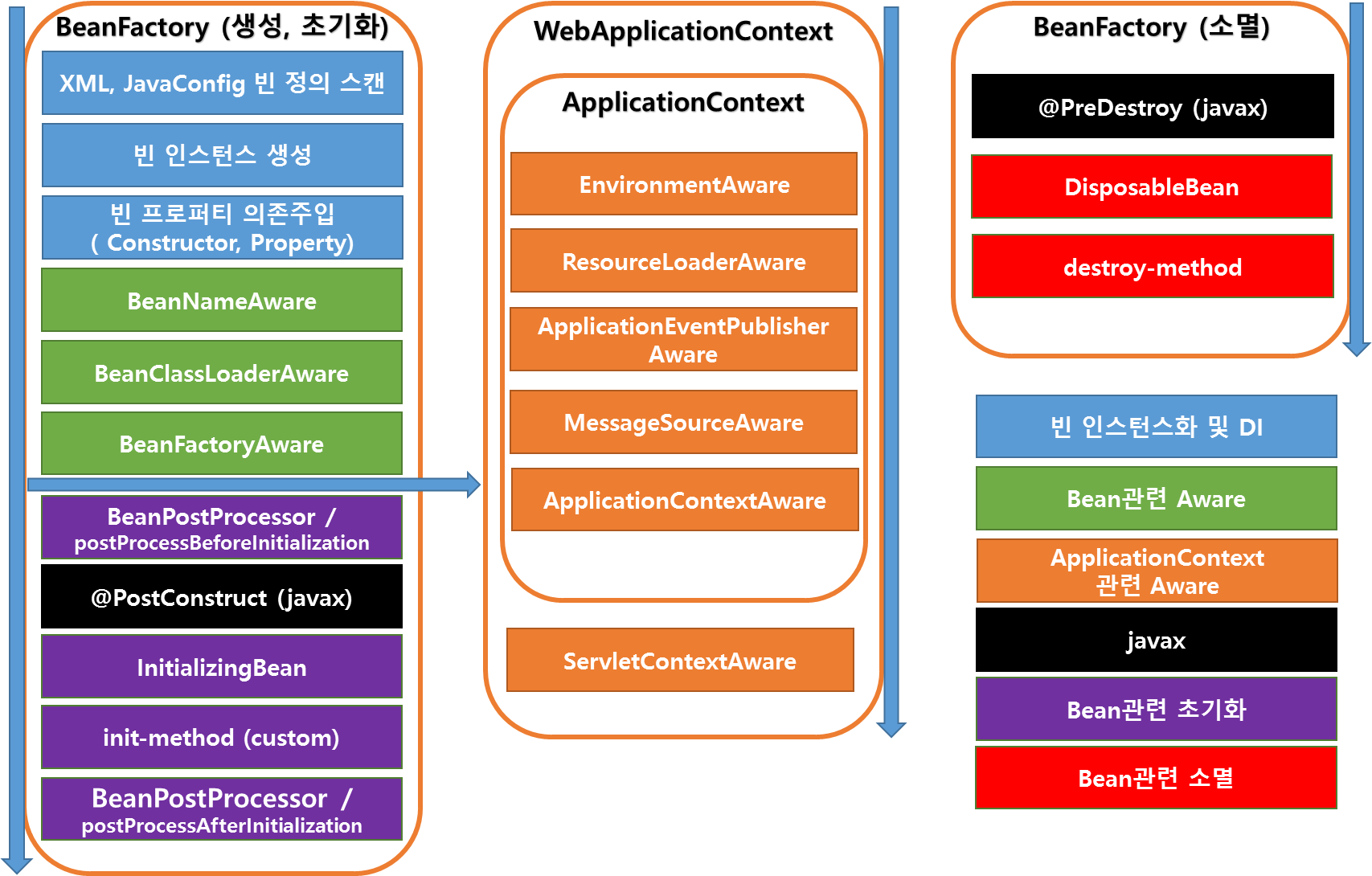
}

🡺여기서 무엇을 느꼈는가? 실제 ApplicationContext는 구현체들과 무관하며, 실제 구현 코드는 BeanFactory 구현체들이 담당하며, ApplicationContext는 이 구현체들을 사용할 뿐이다. (위임)

**-DefaultListableBeanFactory (BeanFactory 의 구현체)**

public class DefaultListableBeanFactory {  
 public <T> T getBean(Class<T> requiredType, Object... args) throws BeansException {  
 NamedBeanHolder namedBean = this.resolveNamedBean(requiredType, args);  
 if(namedBean != null) {  
 return namedBean.getBeanInstance();  
 } else {  
 BeanFactory parent = this.getParentBeanFactory();  
 if(parent != null) {  
 return parent.getBean(requiredType, args);  
 } else {  
 throw new NoSuchBeanDefinitionException(requiredType);  
 }  
 }  
 }  
}

**Bean LifeCycle**



**Aware ?**

Bean이 Spring 컨테이너와 리소스에 대한 정보를 얻어 오기 위해 Aware 인터페이스 제공

우리는 단지 setter 메소드만 구현하면 정보를 받아올 수 있다.

**1) 빈 인스턴스화 및 DI**

**-XML, JavaConfig 빈 정의 스캔**

XML 방식

<bean id="alarmDevice" class="mad.spring.ch4.homecontrol.SmsAlarmDevice"/>

JavaConfig 방식

@Configuration  
public class SpringConfig{  
 @Bean  
 public AlarmDevice alarmDevice

return new SmsAlarmDevice();  
 }  
}

**-빈 인스턴스 생성**

좀더 확인

**-빈 프로퍼티에 의존성 주입**

<bean id="Lifecycle" class="com.sjb.core.one.Lifecycle">  
 <constructor-arg value="simjunbo"/> // Constructor  
 <property name="name"> // Setter  
 <value>aiden</value>  
 </property>  
</bean>

**2) BeanFactory의 Aware**

**-BeanNameAware (setBeanName)**

public class BeanNameAwareExample implements BeanNameAware {  
 @Override  
 public void setBeanName(String name) {  
 System.*out*.println("bean name is : " + name);  
 }  
}

스프링에서 관리되는 bean 내부에서 id나 name이 무엇으로 지정되어 있는지 확인이 필요한 경우. 트래킹 로그용…

**-BeanClassLoaderAware (setBeanClassLoader)**

public class BeanClassLoaderAwareExample implements BeanClassLoaderAware {  
 @Override  
 public void setBeanClassLoader(ClassLoader classLoader) {  
 System.*out*.println("parent class loader : "+classLoader.getParent());  
 }  
}

**-BeanFactoryAware (setBeanFactory)**

public class BeanFactoryAwareExample implements BeanFactoryAware {  
 private BeanFactory beanFactory;  
  
 @Override  
 public void setBeanFactory(BeanFactory beanFactory) throws BeansException {  
 this.beanFactory = beanFactory;  
 }  
}

어떤 클래스에서 bean을 꺼낼(look up) 수 있는factory가 필요하다면 사용.

한마디로 bean 정보 꺼내올 때 사용.

**3) ApplicationContext의 Aware**

**-EnvironmentAware (setEnvironment)**

public class EnvironmentAwareExample implements EnvironmentAware {  
 @Override  
 public void setEnvironment(Environment environment) {  
   
 }  
}

**-ResourceLoaderAware (setResourceLoader)**

public class ResourceLoaderAwareExample implements ResourceLoaderAware {  
 @Override  
 public void setResourceLoader(ResourceLoader resourceLoader) {

...  
 }  
}

**-ApplicationEventPublisherAware (setApplicationEventPublisher)**

public class ApplicationEventPublisherAwareExample implements ApplicationEventPublisherAware {  
 @Override  
 public void setApplicationEventPublisher(ApplicationEventPublisher event) {

...  
 }  
}

**-MessageSourceAware (setMessageSource)**

public class MessageSourceAwareExample implements MessageSourceAware {  
 @Override  
 public void setMessageSource(MessageSource messageSource) {  
 ...  
 }  
}

**-ApplicationContextAware (setApplicationContext)**

public class ApplicationContextAwareExample implements ApplicationContextAware {  
 @Override  
 public void setApplicationContext(ApplicationContext context) throws BeansException {

...  
 }  
}

빈이 실행되는 환경인 ApplicationContext 인스턴스에 접근할 수 있다.

이것도 bean 정보가 필요 할 때 사용하는듯.. 다른점은 BeanFactory와 ApplicationContext 차이?

**-ServletContextAware (setServletContext)**

public class ServletContextAwareExample implements ServletContextAware {  
 @Override  
 public void setServletContext(ServletContext servletContext) {  
 ...

}  
}

**4)Bean관련 초기화**

**-BeanPostProcessor (postProcessBeforeInitialization)**

public class BeanPostProcessorExample implements BeanPostProcessor {  
  
 @Override  
 public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
 if("sample".equals(beanName)){  
 Sample sample = (Sample) bean;  
 sample.test2();  
 }  
 return bean;  
 }  
}

ApplicationContext이 관리하는 모든 빈에 대해서 Initializing 전/후 시점으로 수행할 프로세스 등록

**-@PostConstruct (JSR-250)**

public class PostConstructExample {  
 String message;  
  
 public String getMessage() {  
 return message;  
 }  
  
 public void setMessage(String message) {  
 this.message = message;  
 }  
  
 @PostConstruct  
 public void initIt() throws Exception {  
 System.*out*.println("@PostConstruct : " + message);  
 }  
}

@PostConstruct를 이용하면 Bean 객체를 생성한 이후에 초기화를 수행할 메소드를 지정할 수 있다.

**-InitializingBean (afterPropertiesSet)**

public class InitializingBeanExample implements InitializingBean {  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 ...

}  
}

**-init-method**

// XML

<bean id="InitMethod" class="com.sjb.core.InitMethodExample" init-method="init"/>

// Java

public class InitMethodExample {  
 public void init() {  
 System.*out*.println("init-method");  
 }  
}

**-BeanPostProcessor (postProcessAfterInitialization)**

public class BeanPostProcessorExample implements BeanPostProcessor {  
  
 @Override  
 public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
 if("initializingBean".equals(beanName)){  
 System.*out*.println("BeanPostProcessor - postProcessAfterInitialization");  
/\* Sample sample = (Sample) bean;  
 sample.test3();\*/  
 }  
 return bean;  
 }  
}

**5)Bean관련 소멸**

**- @PreDestory**

public class PreDestroyExample {  
 @PreDestroy  
 public void cleanUp() throws Exception {

...  
 }  
}

@PreDestory는 Bean 객체의 라이프 사이클이 종료되기 직전에 사용한 자원의 해제할 필요가 있을 경우등에 사용.

-**DisposableBean (destroy)**

public class DisposableBeanExample implements DisposableBean {  
  
 @Override  
 public void destroy() throws Exception {  
 System.*out*.println("destroy");  
 }  
}

**-destroy-method**

// XML

<bean id="destroyMethod" class="com.sjb.core.InitMethodExample" destroy-method

="destroyMethod"/>

// Java

public class DestroyMethodExample {  
 public void destroyMethod() {  
 System.*out*.println("destroy-method");  
 }  
}