

# Spring Object Lifecycle

**Bok, Jong Soon**  
**javaexpert@nate.com**  
**<https://github.com/swacademy/Spring5>**

# Spring Bean Lifecycle

## ■ Spring Container 생성

```
GenericXmlApplicationContext context =  
    new GenericXmlApplicationContext();
```

## ■ Spring Container 설정

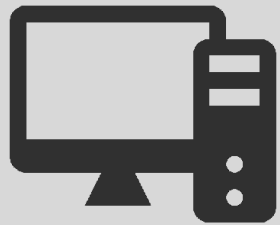
```
context.load("classpath:applicationContext.xml");  
context.refresh(); //생성자를 사용하지 않을 때는 반드시 refresh() 할 것
```

## ■ Spring Container 사용

```
Student student = context.getBean("student", Student.class);  
System.out.println(student);
```

## ■ Spring Container 종료

```
context.close();
```



**THE LAB**

# Task 1. Lab



# Spring Bean Lifecycle (Cont.)

- `context.refresh()` 할 때 bean 초기화 과정에서는 `InitializingBean` Interface의 `afterPropertiesSet()` 또는 `@PostConstruct` Annotation 을 호출
  - `ctx.load("classpath:applicationContext.xml")` ; 에서 Bean이 초기화 되고,
  - `ctx.refresh()` 할 때 Bean이 생성된다.
  - 즉 Bean이 초기화될 때 `afterPropertiesSet()` 이 호출된다.
- `context.close()` 할 때 bean 소멸 과정에서는 `DisposableBean` interface의 `destroy()` 또는 `@PreDestroy()` 를 호출
- 만일 `close()` 할 때 Container가 소멸되면 그 안의 모든 Bean은 자동 소멸 된다.
  - 반면 Bean만 소멸하고자 한다면 `student.destroy()` 를 이용하면 된다.

## Spring Bean Lifecycle (Cont.)

- JSR 250(Java Platform용 공용 Annotation)은 다양한 Java 기술에서 사용되는 표준 Annotation을 정의한다.
- **@PostConstructor**와 **@PreDestroy** 역시 JSR 250에 속해 있다.
  - <http://jcp.org/en/jsr/detail?id=250> 참조



**THE LAB**

## Task 2. Lab



# Spring Bean Scope

- Spring Container가 생성되고, Spring Bean이 생성될 때, 생성된 Spring Bean은 Scope를 가지고 있다.
- Scope란 쉽게 생각해서 해당하는 객체가 어디까지 영향을 미치는지 결정하는 것이라고 생각하면 된다.
  - **singleton** : Spring Container에 단 한 개의 Bean 객체만 존재, 기본값
  - **prototype** : Bean을 사용할 때마다 객체를 생성
  - **request** : Http 요청마다 Bean 객체를 생성, **WebApplicationContext**에서만 적용 가능
  - **session** : Http Session 마다 Bean 객체를 생성, **WebApplicationContext**에서만 적용 가능



**THE LAB**

## Task 3. Lab





## <import> or @Import

- Spring 기반의 Application은 단순한 <bean> 등록 외에도 Transaction관리, 예외처리, 다국어 처리 등 복잡하고 다양한 설정이 필요하다.
- 이런 모든 설정을 하나의 파일로 모두 처리할 수도 있지만, 그렇게 하면 Spring 설정 파일이 너무 길어지고 관리가 어려워진다.
- 결국, 기능별 여러 XML 파일로 나누어 설정하는 것이 더 효율적인데, 이렇게 분리하여 작성한 설정 파일들을 하나로 통합할 때 <import>를 사용.
- Java Annotation 기반에서는 @Import를 사용한다.

## <import> or @Import (Cont.)

- context-datasource.xml

```
<beans>
```

```
...
```

```
</beans>
```

- context-transaction.xml

```
<beans>
```

```
...
```

```
</beans>
```

- 이럴 경우, 2개의 XML 설정 File을 하나로 통합할 수 있다.

```
<beans>
```

```
  <import resource="context-datasource.xml" />
```

```
  <import resource="context-transaction.xml" />
```

```
</beans>
```

## <import> or @Import (Cont.)

- AppConfig.java
- AppConfig1.java
- 이럴 경우, 2개의 Configuration Java File을 하나로 통합할 수 있다.

```
@Configuration
@Import({AppConfig.class, AppConfig1.class})
public class ApplicationConfig {

}
```

# <bean>의 기타 속성들

## ■ **init-method** 속성

- Servlet Container는 web.xml 파일에 등록된 Servlet 클래스의 객체를 생성할 때 기본생성자만 인식한다.
- 따라서 생성자로 Servlet 객체의 멤버변수를 초기화할 수 없다.
- 그래서 Servlet에서는 **init()**를 재정의하여 멤버변수를 초기화할 수 있다.
- Spring Container 역시 Spring 설정 파일에 등록된 Class를 객체 생성할 때 기본생성자를 호출한다.
- 따라서 객체를 생성할 때 멤버변수를 초기화하는 작업이 필요하다면, Servlet의 **init()** 같은 Method가 필요하다.
- 이를 위해 Spring에서는 <bean>에 **init-method** 속성을 지원한다.

## <bean>의 기타 속성들 (Cont.)

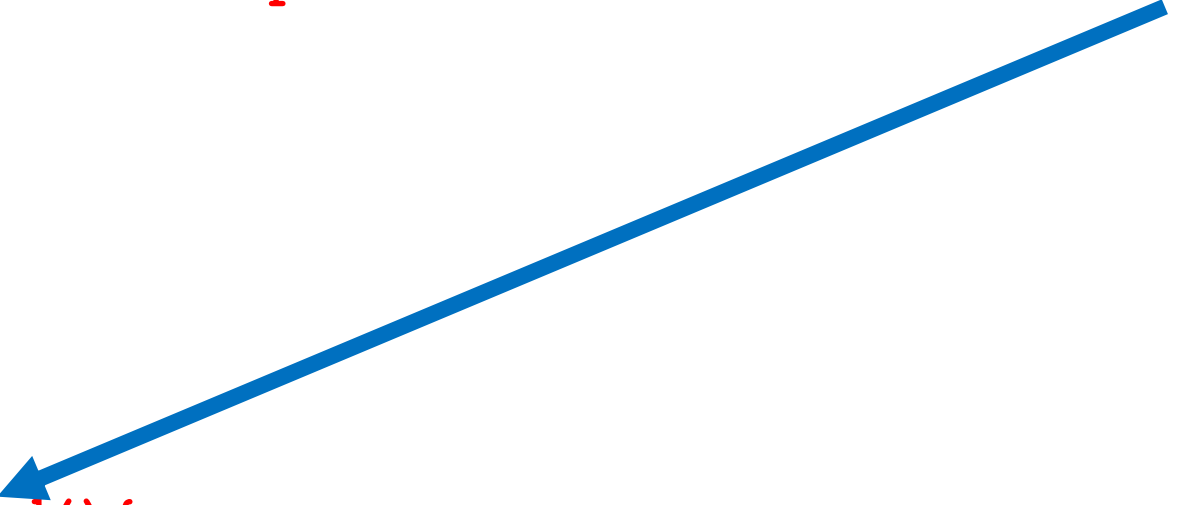
### ■ `init-method` 속성

```
<bean id="hello" class="com.example.Hello" init-method="initMethod" />
```

Hello.java

```
package com.example;
```

```
public class Hello {  
    public void initMethod() {  
        System.out.println("초기화 작업");  
    }  
}
```



## <bean>의 기타 속성들 (Cont.)

### ■ `init-method` 속성

- Spring Container는 `<bean>`에 등록된 Class 객체를 생성한 후에 `init-method` 속성으로 지정된 `initMethod()` Method를 호출한다.
- 이 Method는 Member변수에 대한 초기화를 처리한다.

## <bean>의 기타 속성들 (Cont.)

### ■ **destroy-method** 속성

- **init-method**와 마찬가지로 <bean>에서 **destroy-method** 속성을 이용하여 Spring Container가 객체를 삭제하기 전에 호출될 임의의 Method를 지정할 수 있다.
- Container가 종료되기 직전에 Container는 자신이 관리하는 모든 객체를 삭제하는데, 이때 **destroy-method** 속성으로 지정한 **destroyMethod()** Method는 객체가 삭제되기 직전에 호출된다.

## <bean>의 기타 속성들 (Cont.)

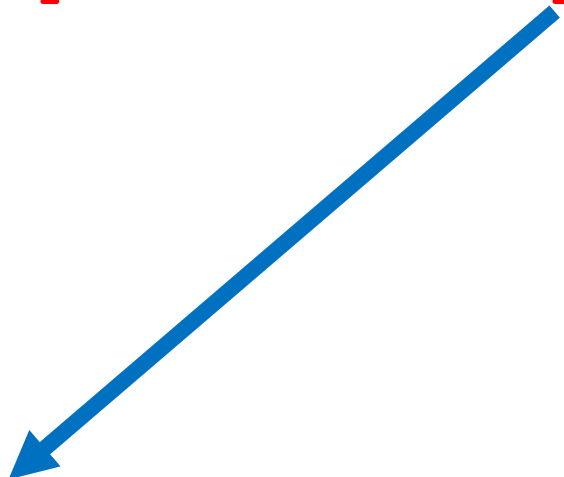
### ■ **destroy-method** 속성

```
<bean id="hello" class="com.example.Hello"  
      destroy-method="destroyMethod" />
```

Hello.java

```
package com.example;
```

```
public class Hello {  
    public void destroyMethod() {  
        System.out.println("객체 삭제 처리 작업");  
    }  
}
```





## <bean>의 기타 속성들 (Cont.)

### ■ **lazy-init** 속성

- **ApplicationContext**를 이용하여 Container를 구동하면 Container가 구동되는 시점에 Spring 설정 파일에 등록된 **<bean>** 들을 생성하는 즉시 로딩(pre-loading) 방식으로 동작한다.
- 즉, **singleton** 범위의 Bean은 기본적으로 사전(Pre) Instance化되므로 Spring Container의 Instance가 생성될 때 **singleton** 범위 Bean의 Instance도 함께 생성
- 그런데 어떤 Bean은 자주 사용되지 않으면서 Memory를 많이 차지하여 System에 부담을 주는 경우도 있다.
- 따라서 Spring 에서는 Container가 구동되는 시점이 아닌 해당 Bean이 사용되는 시점에 객체를 생성하도록 **lazy-init="true"**로 설정하면 Spring Container는 해당 Bean을 미리 생성하지 않고 Client가 요청하는 시점에 생성한다.
- 결국, Memory 관리를 더 효율적으로 할 수 있게 해 준다.

```
<bean id="lazyBean" class="example.lazyBean"
```

```
    lazy-init="true" />
```