Crucial Namespaces 'p' and 'c'

Namespace 'p'

In spring**, p-namespace** is the XML short cut to inject dependency in bean. **p-namespace replaces <property> tag of XML**. p-namespace has no XSD definition and it only exits in core of spring. We can directly assign the attribute name of the class with p-namespace within bean tag. We can use p-namespace in place of <property> tag in spring XML. It is easier and clear to use that will **increase readability** of spring XML application context. Suppose we have <bean> definition in XML as follows.

```
<bean id="address" class="com.soprasteria.training.model.Address"
          p:houseNo="C-123/A" p:street="Street Number 6" p:city="New
Delhi" p:state="Delhi" />

<bean id="user" class="com.soprasteria.training.model.User"
          p:id="101" p:name="Rohan" p:email="rohan@gmail.com"
          p:address-ref="address" />
```

namespace 'c'

c-namespace has been introduced in spring 3.1. It replaces the old style of constructor-arg. The bean which needs to be configured with c-namespace, must have constructor to accept those arguments.

```
<bean id="address" class="com.soprasteria.training.model.Address"
          c:houseNo="C-123/A" c:street="Street Number 6"
          c:city="New Delhi" c:state="Delhi" />

     <bean id="user" class="com.soprasteria.training.model.User"
          c:id="101" c:name="Rohan" c:email="rohan@gmail.com"
          c:address-ref="address" />
```

Spring Collection Injection (List, Set, Map and Properties)

We can also inject Collection in Spring container. The collections are List, Set, Map and Properties. To populate values in the collection, spring provides different tags as follows.

1. <list> is for List
2. <set> is for Set
3. <map> is for Map
4. <props> is for Properties

Spring Collection List Injection

Spring provides <list> tag to inject java List using spring application context XML. <list> is used within <property> or <constructor-arg>. To add values in the List, spring provides <value> that is nested within <list>.

```
<property name="list">
       <list>
            <value>List 1. Java</value>
            <value>List 2. Python</value>
       </list>
 </property>
```

Spring Collection Set Injection

Spring provides <set> tag to inject java Set using spring application context XML. <set> is used within <property> or <constructor-arg>. To add values in the Set, spring provides <value> that is nested within <set>.

```
 <property name="set">
         <set>
           <value>Set 1. IT Employees</value>
           <value>Set 2. Test Engineers</value>
         </set>
  </property>
```

Spring Collection Map Injection

Spring provides <map> tag to inject java Map using spring application context XML. <map> is used within <property> or <constructor-arg>. To add values in the Map, spring provides <entry> that is nested within <map>.

```
     <property name="map">
                <map>
                     <entry key="0" value="Map(Emails)" />
                     <entry key="1" value="Map(Account Numbers)" />
                </map>
    </property>
```

Spring Collection Properties Injection

Spring provides <props> tag to inject java Properties using spring application context XML. <props> is used within <property> or <constructor-arg>. To add values in the Properties, spring provides <prop> that is nested within <props>.

```
       <property name="properties">
                <props>
                     <prop key="propKeyA">This is Property-1</prop>
                <prop key="propKeyB">This is Property-2</prop>
            </props>
        </property>
```

Collection Injection with <ref>

To inject collection of objects to our beans can also be achieved by <ref>. We can provide reference of bean in our collection element population.

```
        <property name="addressList">
            <list>
                <ref bean="address1" />
                <ref bean="address2" />
            </list>
        </property>
```

Person.java

```
public class Person {

    private List<String> list;
    private Set<String> set;
    private Map<Integer, String> map;
    private Properties properties;
    private List<Address> addressList;
  //getter Setter , constructors

}
```

beans.xml

```
<bean id="address1" class="com.soprasteria.training.model.Address"
        c:houseNo="C-123/A" c:street="Street Number 6" c:city="New Delhi"
        c:state="Delhi" />
    <bean id="address2" class="com.soprasteria.training.model.Address"
        c:houseNo="A-1234" c:street="Street Number 1" c:city="Noida"
        c:state="Uttar Pradesh" />

    <bean id="person" class="com.soprasteria.training.model.Person">
        <property name="list">
            <list>
                <value>List 1. Java</value>
                <value>List 2. Python</value>
            </list>
        </property>
        <property name="set">
            <set>
                <value>Set 1. IT Employees</value>
                <value>Set 2. Test Engineers</value>
            </set>
        </property>
        <property name="map">
              <map>
                    <entry key="0" value="Map(Emails)" />
                    <entry key="1" value="Map(Account Numbers)" />
              </map>
        </property>
        <property name="properties">
              <props>
                    <prop key="propKeyA">This is Property-1</prop>
                    <prop key="propKeyB">This is Property-2</prop>
              </props>
        </property>
        <property name="addressList">
              <list>
                    <ref bean="address1" />
                    <ref bean="address2" />
              </list>
        </property>
  </bean>
```

SpringContainer.java

```
        ApplicationContext context = new
  FileSystemXmlApplicationContext("beans.xml");
        Person person = (Person)context.getBean("person");
        //access list
     System.out.println("---access list---");
     List<String> list=person.getList();
     System.out.println(list.get(0));
     System.out.println(list.get(1));
     //access set
     System.out.println("---access set---");
     Set<String> set=person.getSet();
     Iterator<String> itr= set.iterator();
     while(itr.hasNext()){
        System.out.println(itr.next());
     }
     //access map
     System.out.println("---access map---");
     Map<Integer,String> map=person.getMap();
     System.out.println(map.get(0));
     System.out.println(map.get(1));
     //access properties
     System.out.println("---access properties---");
     Properties prop=person.getProperties();
     System.out.println(prop.getProperty("propKeyA"));
     System.out.println(prop.getProperty("propKeyB"));
     //Access Book List
     System.out.println("---Access Address List---");
     System.out.println(person.getAddressList().get(0).getHouseNo());
     System.out.println(person.getAddressList().get(1).getState());
```

Collection Merging in Spring

In spring, java collections can be injected in bean using application xml. According to the need, we can merge collection data while creating bean. **There will be a parent and child bean**. **Child will inherit the parent data**. **It is done by the attribute merge="true".** This attribute is available in all collection tags like <list/>, <set/>, <map/> and <props/> . The child bean will use parent attribute to inherit parent bean data. The child bean collection tag will use merge="true" attribute to inherit parent collection elements. **When we access child bean, we will get elements of child as well as parent bean collection**. This is collection merging of beans in spring.

Collection Merging using merge="true" using XML

collection-merging.xml:

```
<bean id="parentCollection"
class="com.soprasteria.training.model.CollectionMerge">
       <property name="set">
      <set>
            <value>Parent Class Set-1</value>
            <value>Parent Class Set-2</value>
        </set>
       </property>
    </bean>
    <bean id="childCollection" parent="parentCollection">
       <property name="set">
      <set merge="true">
            <value>Child Class Set-1</value>
            <value>Child Class Set-2</value>
        </set>
       </property>
    </bean>
```

CollectionMerge.java

```
public class CollectionMerge {
      private Set<String> set;

      public Set<String> getSet() {
            return set;
      }
      public void setSet(Set<String> set) {
            this.set = set;
      }
}
```

CollectionMergeContainer.java

```
            ApplicationContext context = new
            FileSystemXmlApplicationContext("collection-merging.xml");
            CollectionMerge parent = (CollectionMerge)
            context.getBean("parentCollection");
```

```
        // Access Parent Collection
        System.out.println("---Elements in parent bean---");
        Set<String> parentSet = parent.getSet();
        parentSet.forEach(System.out::println);
        CollectionMerge child = (CollectionMerge)
        context.getBean("childCollection");
        // Access Child Collection
        System.out.println("---Elements in child bean---");
        Set<String> childSet = child.getSet();
        childSet.forEach(System.out::println);
```

Spring - Property Editors

Spring core framework uses PropertyEditor instances to convert text to object and vice versa, wherever it is needed.

The PropertyEditor concept is part of the JavaBeans specifications.

https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/java/beans/PropertyEditor.html

PropertyEditor was originally designed to be used in **Swing applications**. JavaBeans specification defines API to introspect and extract the bean inner details which can be used to show bean properties visually as components and edit them by using PropertyEditors.

Following are the two of the cases where Spring uses PropertyEditor(s).

1. Setting Bean properties defined in XML configuration files. **As all values of bean properties are in text format there,** Spring needs to convert them into Java objects by using an appropriate PropertyEditor instance. For example if the target bean's 'property' is the java.lang.Class type then Spring uses ClassEditor (a Spring implementation of PropertyEditor).
2. In Spring MVC framework, all incoming HTTP request parameters and other information is in text, that's where PropertyEditors are also used for Java Object conversions.

Spring default PropertyEditors

Spring provides many default editors, e.g. ClassEditor, FileEditor, LocaleEditor, CurrencyEditor etc.

Link: https://docs.spring.io/spring/docs/1.2.x/reference/validation.html

Here's the CurrencyEditor code snippet to understand the concept better:

```
public class CurrencyEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        super.setValue(Currency.getInstance(text));
```

```java
	}

	@Override
	public String getAsText() {
		Currency value = (Currency) super.getValue();
		return (value != null ? value.getCurrencyCode() : "");
	}
}
```

When spring parses following XML config, it first calls the method setAsText(..) and then getValue() to inject the Currency instance in the bean.

```xml
<bean id="currency" class="com.soprasteria.training.model.CurrencyBean"
p:currency="USD" />
```

Custom PropertyEditors Spring

In spring, when we inject setter value as string, internally spring uses build in PropertyEditors to change that string to actual object. According to the need, we can create custom PropertyEditors that will create required object by given injected value. Lets understand step by step. Find the beans Customer and CustomerType.

Customer.java

```java
public class Customer {
	private String name;
	private CustomerType type;
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public CustomerType getType() {
		return type;
	}
	public void setType(CustomerType type) {
		this.type = type;
	}

}
```

CustomerType.java

```java
public class CustomerType {
	private String typeName;
	public CustomerType() {
	}

	public CustomerType(String typeName) {
		super();
		this.typeName = typeName;
	}
	public String getTypeName() {
		return typeName;
	}

	public void setTypeName(String typeName) {
		this.typeName = typeName;
	}
}
```

Customer class has a property of CustomerType. Now find the XMl code which is injecting value to Customer. For the variable type, we need CustomerType Object, but we have injected String i.e Regular.

```
<bean id="customer"
		class="com.soprasteria.training.model.Customer" p:name="Roger"
p:type="Regular" />
```

Now Here is the of custom PropertyEditors. Find the custom property editors below.

```
public class CustomerTypeEditor extends PropertyEditorSupport {

	@Override
	public void setAsText(String text) throws IllegalArgumentException {
		setValue(new CustomerType(text.toUpperCase()));
	}

}
```

PropertyEditorSupport in Spring

To create custom PropertyEditors, we need to extend PropertyEditorSupport and need to override setAsText method to create required object from string. Now we need to register this CustomerTypeEditor in spring to work. Find the XML code.

```
<bean id="customer"
		class="com.soprasteria.training.model.Customer"
p:name="Roger" p:type="Regular" />

<bean
class="org.springframework.beans.factory.config.CustomEditorConfig
urer">
	<property name="customEditors">
	 <map>
	   <entry key="com.soprasteria.training.model.CustomerType"
value="com.soprasteria.training.model.CustomerTypeEditor"/>
	 </map>
```

```
        </property>
    </bean>
```

CustomEditorConfigurer in Spring

CustomEditorConfigurer has a collection property customEditors that takes custum PropertyEditors. Now run the main method to check the output. What we have done is that CustomerTypeEditor will return CustomerType object with Upper case property value.

```
ApplicationContext context = new
FileSystemXmlApplicationContext("property-editor.xml");
Customer customer = (Customer)context.getBean("customer");
CustomerType customerType = customer.getType();
//Will return Customer type in Capital Letters
System.out.println(customerType.getTypeName());
```

Spring Factory Method

Normally, Spring instantiates a class and performs dependency injection. However, sometimes it may be necessary to **instantiate a class via another class** (usually called a Factory class). In such a scenario, Spring should not create the class on its own but simply delegate the instantiation to the Factory class.

- ❖ Spring provides a way to delegate class instantiation to another class by using factory-method attribute of bean tag.
- ❖ Essentially, there is a static method defined in the Factory class that creates the instance of the required bean, hence the name factory-method.
- ❖ Typically, factory-method is used in application integration where objects are constructed in a more complex way using a Factory class.
- ❖ Also, there may be third party libraries that may need to be used within Spring and these third party libraries use Factory classes to instantiate other classes. In such case, the use of factory-method attribute greatly simplifies integrability of Spring based applications with third party libraries.

We will be using **factory-method** and **factory-bean** attribute in our configuration for the Injection of Bean.

factory-method: factory-method is the method that will be invoked while injecting the bean. It is used when the factory method is static
factory-bean: factory-bean represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

Factory Method Types

There can be three types of factory method:

1. A static factory method that returns instance of its own class. It is used in singleton design pattern.

```
    <bean id="employee"
          class="com.soprasteria.training.model.EmployeeFactory"
          factory- method="createEmployee" />
```

2. A static factory method that returns instance of another class. It is used instance is not known and decided at runtime.

```
    <bean id="employee"
          class="com.soprasteria.training.model.EmployeeFactory"
           factory-method="createEmployee">
          <constructor-arg value="director" />
    </bean>
```

3. A non-static factory method that returns instance of another class. It is used instance is not known and decided at runtime.

Singleton Design pattern Factory Method

Employee.java

```
public class Employee {

      private Integer id;
      private String firstName;
      private String lastName;
      private String designation;
      //Default & parametrized constructors
      //Getters & Setters & toString()
}
```

EmployeeFactory.java for Singleton Design pattern

```
public class EmployeeFactory {
      private static final Employee EMPLOYEE = new Employee();
      // Private Constructor
      private EmployeeFactory() {
            System.out.println("Employee Private Constructor");

      }
      public static Employee createEmployee() {

            System.out.println("Factory Method");
            return EMPLOYEE;
      }
}
```

factory-methods.xml

```
<bean id="employee"
          class="com.soprasteria.training.model.EmployeeFactory"
    factory-method="createEmployee">
  </bean>
```

FactoryContainer.java

```
ApplicationContext context = new FileSystemXmlApplicationContext("factory-
methods.xml");
          Employee employee = (Employee)context.getBean("employee");
          System.out.println(employee);
```

Factory Method with static methods:

Employee.java(Same as in first Step)

EmployeeFactory.java

```
public class EmployeeFactory {
     //Private Constructor
     private EmployeeFactory() {

     }
     public static Employee createEmployee(String type)
    {
        if ("manager".equals(type) || "director".equals(type))
        {
            Employee employee = new Employee();

            employee.setId(-1);
            employee.setFirstName("Sample");
            employee.setLastName("Sample");
            //Set designation here
            employee.setDesignation(type);

            return employee;
        }
        else
        {
            throw new IllegalArgumentException("Unknown product");
        }
    }
}
```

factory-methods.xml

```
      <bean id="employee"
          class="com.soprasteria.training.model.EmployeeFactory"
          factory-method="createEmployee">
          <constructor-arg value="director"></constructor-arg>
     </bean>
```

FactoryContainer.java(Same as in first Step)

Spring Instance Factory Method(non-static methods)

Employee.java(Same as above)

EmployeeFactory.java

```java
public class EmployeeFactory {

	// Private Constructor
	private EmployeeFactory() {

	}
	public static EmployeeFactory createEmployee() {
		return new EmployeeFactory();
	}
	public Employee getDeveloper() {
		Employee employee = new Employee();
		employee.setId(101);
		employee.setFirstName("Vimal");
		employee.setLastName("Verma");
		employee.setDesignation("Developer");
		return employee;
	}

	public Employee getDesigner() {
		Employee employee = new Employee();
		employee.setId(102);
		employee.setFirstName("Rahul");
		employee.setLastName("Sharma");
		employee.setDesignation("Designer");
		return employee;
	}
}
```

factory-methods.xml

```xml
<bean id="employee"
		class="com.soprasteria.training.model.EmployeeFactory"
		factory-method="createEmployee">
	</bean>
	<bean id="developer"
		class="com.soprasteria.training.model.Employee"
		factory-bean="employee" factory-method="getDeveloper">
	</bean>
	<bean id="designer"
		class="com.soprasteria.training.model.Employee"
		factory-bean="employee" factory-method="getDesigner">
	</bean>
```

FactoryContainer.java

```
ApplicationContext context = new
FileSystemXmlApplicationContext("factory-methods.xml");

Employee developer = (Employee) context.getBean("developer");

System.out.println("**** Developer Details ****");
System.out.println("Employee ID : " + developer.getId());
System.out.println(" First Name : " + developer.getFirstName());
System.out.println("Last Name  : " + developer.getLastName());
System.out.println("Designation : " + developer.getDesignation());

// Get the Employee(Designer) class instance
Employee designer = (Employee) context.getBean("designer");

System.out.println("**** seniormanager Details ****");
System.out.println("Employee ID : " + designer.getId());
System.out.println(" First Name : " + designer.getFirstName());
System.out.println("Last Name  : " + designer.getLastName());
System.out.println("Designation : " + designer.getDesignation());
```

Spring bean scopes

The core of spring framework is it's bean factory and mechanisms to create and manage such beans inside Spring container. The beans in spring container can be created in five scopes i.e. singleton, prototype, request, session and global-session. They are called spring bean scopes.

1. **Singleton**
   This bean scope is default and it enforces the container to have **only one instance per spring container** irrespective of how much time you request for its instance. This singleton behavior is maintained by bean factory itself.

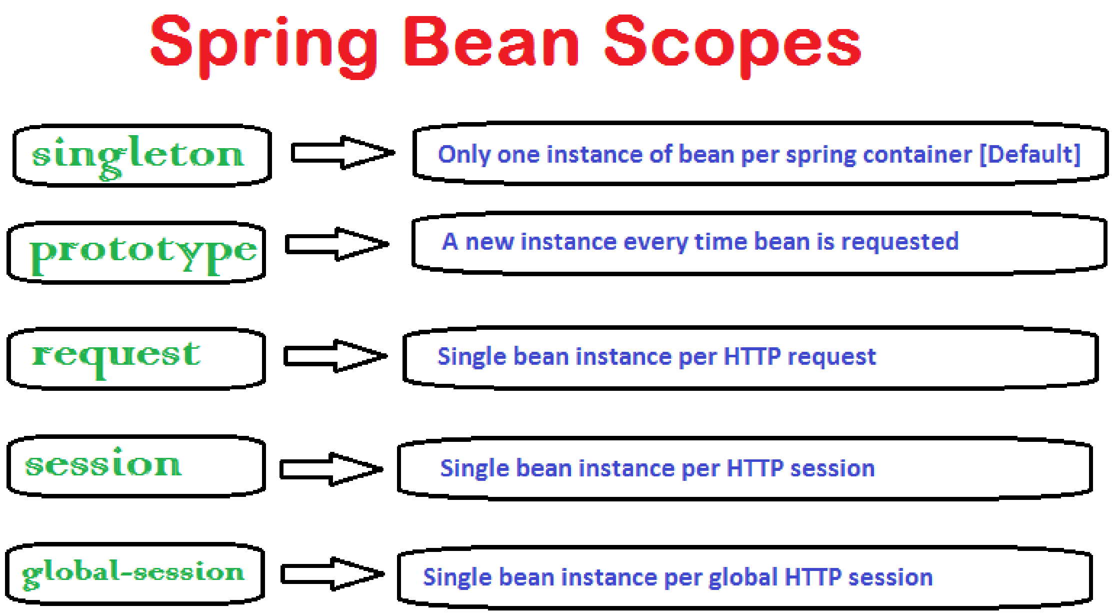
2. **Prototype**
   This bean scope just reverses the behavior of **singleton scope and produces a new instance each and every time a bean is requested**.

Remaining **three bean scopes are web applications related**. Essentially these are available through web aware application context (e.g. WebApplicationContext). Global-session is a little different in sense that it is used when application is portlet based. In portlets, there will be many applications inside a big application and a bean with scope of 'global-session' will have only one instance for a global user session.

3. **request**
   With this bean scope, a new bean instance will be created for each web request made by client. As soon as request completes, bean will be out of scope and garbage collected.

4. **session**
   Just like request scope, this ensures one instance of bean per user session. As soon as user ends its session, bean is out of scope.

5. **global-session**
   It is something which is connected to **Portlet** applications. When your application works in Portlet container it is built of some amount of portlets. Each portlet has its own session, but if your want to store variables global for all portlets in your application than you should store them in global-session. This scope doesn't have any special effect different from session scope in Servlet based applications.


Spring Bean Scopes
singleton
Only one instance of bean per spring container [Default]
prototype
A new instance every time bean is requested
request
Single bean instance per HTTP request
session
Single bean instance per HTTP session
global-session
Single bean instance per global HTTP session

Bean Scope is being defined in the bean using the "scope" attribute:

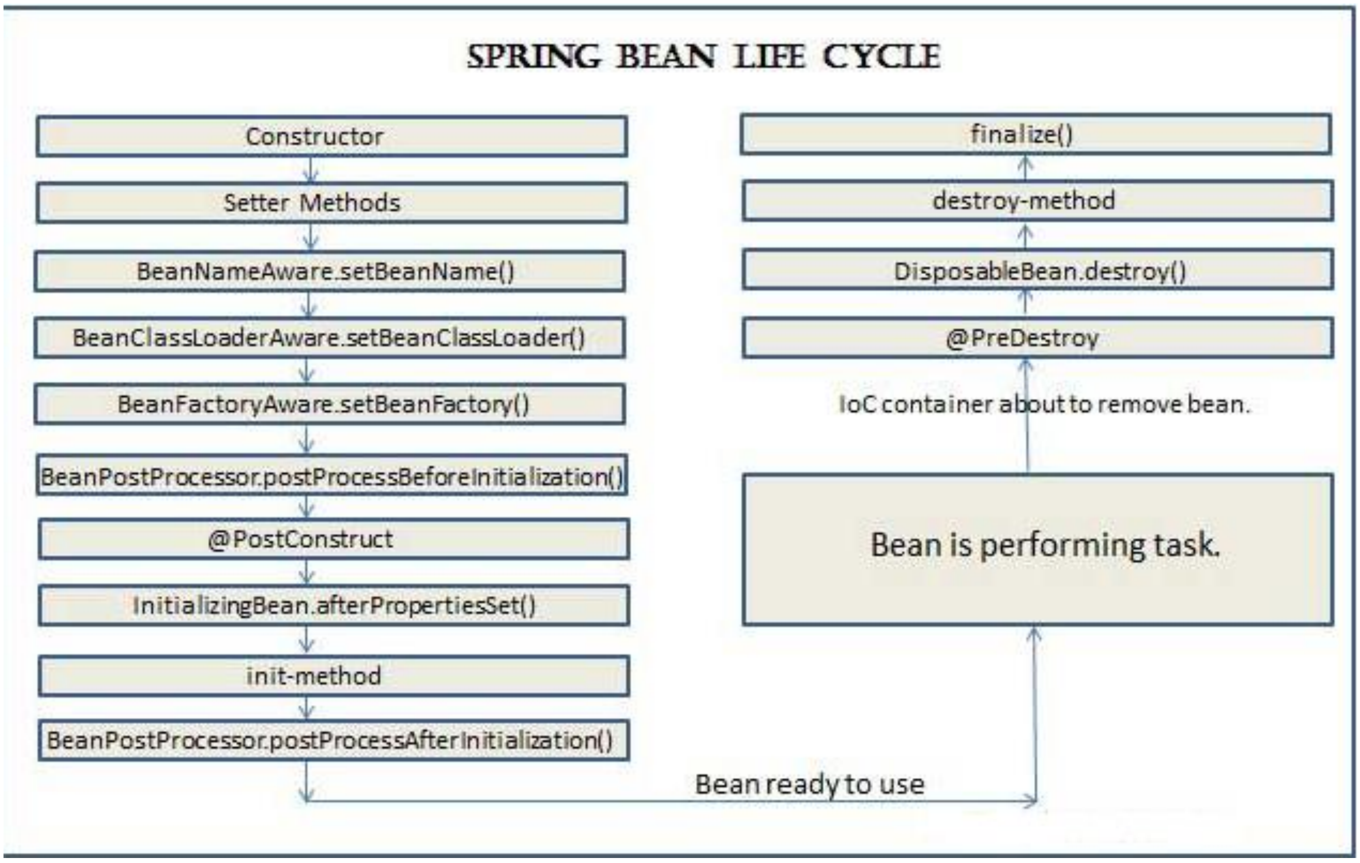
```
<bean id="address"
class="com.soprasteria.training.model.Address"
scope="singleton" />
```

Spring Expression Language SpEL
https://docs.spring.io/spring/docs/3.0.x/reference/expressions.html

Spring Bean Life Cycle

Here we will walk through spring bean life cycle stages. In spring bean life cycle, **initialization** and **destruction** callbacks are involved. **Different spring bean aware classes are also called during bean life cycle**. Once dependency injection is completed, initialization callback methods execute. Their purposes are to check the values that have been set in bean properties, perform any custom initialization or provide a wrapper on original bean etc. **Once the initialization callbacks are completed, bean is ready to be used**. When **IoC** container is about to remove bean, **destruction callback methods execute**. Their purposes are to release the resources held by bean or to perform any other finalization tasks. When more than one initialization and destructions callback methods have been implemented by bean, **then those methods execute in certain order**. Here we will discuss spring bean life cycle step by step. We will discuss the order of execution of initialization and destruction callbacks as well as spring bean aware classes.

Spring Bean Life Cycle Diagram and Steps


SPRING BEAN LIFE CYCLE
Constructor
Setter Methods
BeanNameAware.setBeanName()
BeanClassLoaderAware.setBeanClassLoader()
BeanFactoryAware.setBeanFactory()
BeanPostProcessor.postProcessBeforeInitialization()
@PostConstruct
InitializingBean.afterPropertiesSet()
init-method
BeanPostProcessor.postProcessAfterInitialization()
Bean ready to use
Bean is performing task.
IoC container about to remove bean.
@PreDestroy
DisposableBean.destroy()
destroy-method
finalize()

A bean life cycle includes the following steps.
1. Within IoC container, a spring bean is created using class **constructor**.
2. Now the dependency injection is performed using setter method.
3. Once the dependency injection is completed, **BeanNameAware.setBeanName()** is called. It sets the name of bean in the bean factory that created this bean.
4. Now **BeanClassLoaderAware.setBeanClassLoader()** is called that supplies the bean class loader to a bean instance.

5. Now **BeanFactoryAware.setBeanFactory()** is called that provides the owning factory to a bean instance.
6. Now the IoC container calls **BeanPostProcessor.postProcessBeforeInitialization** on the bean. Using this method a wrapper can be applied on original bean.
7. Now the method annotated with **@PostConstruct** is called.
8. After **@PostConstruct**, the method **InitializingBean.afterPropertiesSet()** is called.
9. Now the method specified by **init-method** attribute of bean in XML configuration is called.
10. And then **BeanPostProcessor.postProcessAfterInitialization()** is called. It can also be used to apply wrapper on original bean.
11. Now the bean instance is **ready to be used**. Perform the task using the bean.
12. Now when the **ApplicationContext shuts down** such as by using **registerShutdownHook()** then the method annotated with **@PreDestroy** is called.
13. After that **DisposableBean.destroy()** method is called on the bean.
14. Now the method specified by **destroy-method** attribute of bean in XML configuration is called.
15. Before garbage collection, **finalize() method of Object is called**.

Initialization Callbacks

In the bean life cycle **initialization callbacks** are those methods which are called just after the properties of the bean has been set by IoC container. The spring **InitializingBean** has a method as **afterPropertiesSet()** which performs **initialization** work after the bean properties has been set. Using **InitializingBean** is not being recommended by spring because it couples the code. We should use **@PostConstruct** or method specified by bean attribute **init-method** in XML which is the same as **initMethod attribute** of **@Bean** annotation in JavaConfig. If all the three are used together, they will be called in below order in bean life cycle.

1. First **@PostConstruct** will be called.
2. Then **InitializingBean.afterPropertiesSet()** is called
3. And then method specified by bean **init-method** in XML or **initMethod of @Bean in JavaConfig.**

Destruction Callbacks

In bean life cycle when a bean is destroyed from the IoC container, **destruction callback** is called. To get the destruction callback, bean should implement spring **DisposableBean interface** and the **method destroy()** will be called. Spring recommends not to use **DisposableBean** because it couples the code**. As destruction callback we should use @PreDestroy annotation** or bean attribute **destroy-method** in XML configuration which is same as **destroyMethod attribute of @Bean in JavaConfig**. If we use all these callbacks together then they will execute in following order in bean life cycle.
1. First **@PreDestroy** will be called.
2. After that **DisposableBean.destroy()** will be called.
3. And then method specified by bean **destroy-method** in XML configuration is called.

Example code for Initialization and Destruction callbacks

Human.java(Bean class)

```
public class Human implements InitializingBean, DisposableBean {
	private String name;
	public Human() {
		System.out.println("Constructor of Human Bean is Invoked!!");
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	// Bean initialization code
	@Override
	public void afterPropertiesSet() throws Exception {
	System.out.println("Initializing method of Human bean is invoked!");
	}
	// Bean destruction code
	@Override
	public void destroy() throws Exception {
		System.out.println("Destroy method of Human bean is invoked!");
	}
}
```

human.xml

```
  <bean name="human" class="com.soprasteria.training.model.Human">
        <property name="name" value="Jason Clarke" />
    </bean>
```

HumanContainer.java

```
public class HumanContainer {
	public static void main(String[] args) {
		// Reading configuration from the spring configuration file.
	ConfigurableApplicationContext context = new
				ClassPathXmlApplicationContext("human.xml");

		Human human = context.getBean("human", Human.class);
		System.out.println("Name= " + human.getName());
		// Closing the context object.
		context.close();
	}
}
```

Output

```
Constructor of Human Bean is Invoked!!
Initializing method of Human bean is invoked!
Name= Jason Clarke
Destroy method of Human bean is invoked!
```

@PostConstruct and @PreDestroy

Here we will discuss the role of JSR-250 **@PostConstruct** and **@PreDestroy** annotation in spring bean life cycle. Spring recommends these annotations to use as initialization and destruction callbacks. **@PostConstruct** annotated method executes just after dependency injection is completed to perform any initialization. It is customary to specify name as **init()**. **@PreDestroy** annotated method executes before the bean is being removed from spring container. It is commonly used to release resources held by bean. It is customary to specify name as **destroy().** This method is called on calling of **close()** method of spring context.

PostConstructBean.java

```
public class PostConstructBean {
	private String name;
	@PostConstruct
	public void init() {
		System.out.println("Inside PostConstruct init()");
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
		System.out.println("---Inside setName---");
	}
	@PreDestroy
	public void destroy() {
		System.out.println("Inside PreDestroy destroy()");
	}
}
```

human.xml

```
	 <bean name="postConstruct"
		class="com.soprasteria.training.model.PostConstructBean"
		init-method="init" destroy-method="destroy">
		<property name="name" value="My Java" />
	</bean>
```

HumanContainer.java

```
ConfigurableApplicationContext context = new
ClassPathXmlApplicationContext("human.xml");
PostConstructBean bean = (PostConstructBean)context.getBean("postConstruct");
System.out.println("Name : "+bean.getName());

// Closing the context object.
context.close();
```

init-method and destroy-method in XML

In spring bean life cycle **init-method and destroy-method** attributes are used to specify **initialization and destruction** callbacks **custom method** respectively in XML configuration. The equivalent attributes **in JavaConfig are initMethod and destroyMethod** respectively in a @Bean annotation. The customary method name for init-method or initMethod is init(). The customary method name for destroy-method or **destroyMethod** is destroy(). init() is called just after bean properties are set. This method is used to perform any custom initialization or to check if the values are set to the properties. destroy() method is called to release any resources before the spring container removes the bean. destroy() will be called before bean is removed from spring container. This method is called on calling of close() method of spring context.

Bean class name

```
public class InitAndDestroy {
	private String name;

	public void init() {
		System.out.println("Inside init() method");
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
		System.out.println("---Inside setName---");
	}

	public void destroy() {
		System.out.println("Inside destroy() method");
	}
}
```

human.xml

```
<bean name="initAndDestroy"
		class="com.soprasteria.training.model.InitAndDestroy"
		init-method="init" destroy-method="destroy">
		<property name="name" value="John Carter" />
</bean>
```

HumanContainer.java

```
ConfigurableApplicationContext context = new
ClassPathXmlApplicationContext("human.xml");

InitAndDestroy bean = (InitAndDestroy)context.getBean("initAndDestroy ");
System.out.println("Name : "+bean.getName());

// Closing the context object.
context.close();
```

Java Configuration

```
@Configuration
public class AppConfig {
  @Bean(name = " initAndDestroy", initMethod="init", destroyMethod="destroy")
  public InitAndDestroy getBean() {
      InitAndDestroy initAndDestroy = new InitAndDestroy ();
      book.setName("John Carter");
      return initAndDestroy;
      }
}
```

Global definition(init-method & destroy-method)

Where as global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag. They are useful when you have a pattern of defining common method names such as init() and destroy() for all your beans consistently. This feature helps you in not mentioning the init and destroy method names for all beans independently.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:c="http://www.springframework.org/schema/c"
      xmlns:p="http://www.springframework.org/schema/p"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd"
      default-init-method="init" default-destroy-method="destroy">


</beans>
```

BeanNameAware

In bean life cycle org.springframework.beans.factory.BeanNameAware interface is aware of bean name in bean factory. This interface needs to be implemented by the bean and the method **setBeanName() of BeanNameAware** should be implemented. **BeanNameAware.setBeanName()** is called just after the dependency injection is completed. Find the sample example.

Book.java

```
public class Book implements BeanNameAware {
      private String bookName;

      @Override
      public void setBeanName(String name) {
            System.out.println("Bean Name:" + name);
      }
```

```
	public String getBookName() {
		return bookName;
	}

	public void setBookName(String bookName) {
		this.bookName = bookName;
	}
}
```

AppConfig.java

```
@Configuration
public class AppConfig {

	@Bean(name = "My Book")
	public Book getBean() {
		Book book = new Book();
		book.setBookName("Spring 5");
		return book;
	}
}
```

BookContainer.java

```
public class BookContainer {
	public static void main(String[] args) {
	AnnotationConfigApplicationContext ctx = new
	AnnotationConfigApplicationContext();
	ctx.register(AppConfig.class);
	ctx.refresh();
	Book book = ctx.getBean(Book.class);
	System.out.println("Book Name:" + book.getBookName());
	ctx.close();
	}
}
```

Notice that Spring loads beans into it's context before we have even requested it. This is to make sure all the beans are properly configured and application fail-fast if something goes wrong.

Information

@Configuration annotation is used to mark a class as a source of the bean definitions. Beans are the components of the system that you want to wire together.
A method marked with the **@Bean annotation** is a bean producer. Spring will handle the life cycle of the beans for you, and it will use these methods to create the beans.

AnnotationConfigApplicationContext : Standalone application context, accepting annotated classes as input - in particular @Configuration-annotated classes, but also plain @Component types and JSR-330 compliant classes using javax.inject annotations. Allows for registering classes one by one using **register(Class) as well as for classpath scanning using scan(String).**

BeanFactoryAware

In bean life cycle org.springframework.beans.factory.BeanFactoryAware interface is implemented by beans when it wants to aware of its owning **BeanFactory**. We need to override **setBeanFactory()** method**. This method is called just after the dependency injection is completed.** Using this method we can change the bean properties value.

Book.java

```
public class Book implements BeanFactoryAware {
	private String bookName;

	public String getBookName() {
		return bookName;
	}

	public void setBookName(String bookName) {
		this.bookName = bookName;
	}

	@Override
	public void setBeanFactory(BeanFactory beanFactory) throws
	  BeansException {
		Book b = beanFactory.getBean(Book.class);
		b.setBookName(getBookName() + "-Updated");
	}
}
```

Rest 2 classes AppConfig and BookContainer will remain Same

BeanPostProcessor

org.springframework.beans.factory.config.**BeanPostProcessor** interface is used for custom modification of newly created bean properties. To use **BeanPostProcessor** we need to create a class and override its two method **postProcessBeforeInitialization()** and **postProcessAfterInitialization()**. In bean life cycle **BeanPostProcessor** is **called before and after initialization callbacks such as InitializingBean.afterPropertiesSet(),** @PostConstruct and init-method. Here in our example we will use **InitializingBean.afterPropertiesSet()** and this will be called between **postProcessBeforeInitialization() and postProcessAfterInitialization()** methods of BeanPostProcessor.

Book.java

```
public class Book implements InitializingBean {
	private String bookName;

	public String getBookName() {
		return bookName;
	}
	public void setBookName(String bookName) {
		this.bookName = bookName;
		System.out.println("---Inside setBookName---");
	}

	@Override
	public void afterPropertiesSet() throws Exception {
		System.out.println("---afterPropertiesSet---");
		bookName = bookName + "-Hello";
	}
}
```

MyBeanPostProcessor.java

```
@Component
public class MyBeanPostProcessor implements BeanPostProcessor {
	@Override
	public Object postProcessBeforeInitialization(Object bean, String
	  beanName) throws BeansException {
   System.out.println("postProcessBeforeInitialization: Bean Name- " +
   beanName);
	if (bean instanceof Book) {
		Book b = (Book) bean;
		b.setBookName(b.getBookName() + "-Before");
	}
	return bean;
}

	@Override
	public Object postProcessAfterInitialization(Object bean, String
		beanName) throws BeansException {
	System.out.println("postProcessAfterInitialization: Bean Name- " +
	   beanName);
	if (bean instanceof Book) {
		Book b = (Book) bean;
		b.setBookName(b.getBookName() + "-After");
	}
	return bean;
}
}
```

And there is no change in BookContainer.java class

Use **@ComponentScan** to make sure that Spring knows about your configuration classes and can initialize the beans correctly. It makes Spring scan the packages configured with it for the @Configuration classes.

Another way to declare a bean is to mark a class with a **@Component** annotation. Doing this turns the class into a Spring bean at the auto-scan time.

You can also use **@Service**, which marks **a specialization of @Component**. Services have no encapsulated state. It tells Spring that it’s safe to manage them with more freedom than regular components.

To wire the application parts together, use the **@Autowired** on the fields, constructors, or methods in a component. Spring’s dependency injection mechanism wires appropriate beans into the class members marked with @Autowired.