Spring – II

Spring Dependency Injection (DI)

Table of Contents

1. Introduction to Dependency Injection (DI)
2. Constructor-based dependency injection
3. Setter-based dependency injection
4. Dependency resolution process
5. Injecting Primitive and String values
6. Injecting Null and Empty String values
7. Injecting Collections(List, Set, Map) and Properties
8. Injecting References of other beans (Collaborators or Dependent Objects)
9. Autowiring collaborators
10. Defining Inner beans
11. Defining Compound Property Names
12. Lazy-initialized beans
13. Bean scopes
14. The singleton scope
15. The prototype scope
16. Singleton beans with prototype-bean dependencies
17. Lifecycle of Bean
18. Lifecycle callbacks
19. Initialization callbacks
20. Destruction callbacks
21. Default initialization and destroy methods
22. Annotation-based container configuration
23. Java-based container configuration

Introduction to Dependency Injection (DI)

1. Dependency injection (DI) is a process whereby objects define their dependencies.
2. These objects are constructed by IoC container or returned by factory methods.
3. The container then injects those dependencies when it creates the bean.
4. This process is fundamentally the inverse, hence the name Inversion of Control (IoC).

There are two ways to define the Dependency injection (DI)-

1. Constructor-based dependency injection
2. Setter-based dependency injection.

Constructor-based dependency injection

1. Constructor-based DI is accomplished by the container by invoking a constructor with a number of arguments.
2. Constructor-based DI does not allow **partial** DI, passing values and no. of parameters must be same.
3. It does not support to inject the dependencies again**(can't re-inject).**
4. Parameters type order and arguments type order must be same.
5. If both orders are different then, you can use index value (start from 0 to no. of parameters).

Setter-based dependency injection

1. Setter-based DI is accomplished by the container by calling **setter** methods, after invoking . a **no-argument** constructor or no-argument static factory method to instantiate your bean.
2. It also supports **partial dependencies** Injection.
3. It supports setter-based DI after some dependencies have already been injected through the setter approach.
4. It supports setter-based DI after some dependencies have already been injected through the constructor approach.
5. **@Required** annotation on a setter method can be used to make the property a required dependency.

NOTE: you can mix **constructor-based** and **setter-based** DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.

Dependency resolution process

The IoC container performs bean dependency resolution as follows:

1. First, the ApplicationContext is created and initialized with configuration metadata that describes all the beans.
2. Configuration metadata can be specified via XML, annotations, or Java code.
3. All the dependencies are expressed in the form of properties and constructor arguments.
4. Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
5. Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument.
6. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

NOTE: CIRCULAR DEPENDENCIES

Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a **BeanCurrentlyInCreationException**.

One possible solution is, avoid constructor injection and use setter injection only.

Injecting Primitive and String values

```
package com.kalibermind.domain;

public class Address {
	private long id;
	private String houseNo;
	private String street;
	private String city;
	private String landmark;
	private String state;
	private long zipcode;

	public Address() {}
	public Address(long id, String houseNo, String street, String city, String landmark, String state, long zipcode) {
		super();
		this.id = id;
		this.houseNo = houseNo;
		this.street = street;
		this.city = city;
		this.landmark = landmark;
		this.state = state;
		this.zipcode = zipcode;
	}

	//Getters and Setters
}
```

Constructor based DI: Parameterized constructor is required.

```
<bean id="address" class="com.kalibermind.domain.Address">
		<constructor-arg value="100" />
		<constructor-arg value="H-1318, Ground Floor" />
		<constructor-arg value="JP Nagar,2nd Phase" />
		<constructor-arg value="Bengaluru" />
		<constructor-arg value="Next to Bangalore Central Mall" />
		<constructor-arg value="Karnataka" />
		<constructor-arg value="560076" />

</bean>
```

Setter based DI: Public no-arguments and setter methods are required.

```
<bean id="address" class="com.kalibermind.domain.Address">
		<property name=”id” value="100" />
		< property name=”houseNo” value="H-1318, Ground Floor" />
		< property name=”street” value="JP Nagar,2nd Phase" />
```

```
                < property name=”city” value="Bengaluru" />
                < property name=”landmark” value="Next to Bangalore Central Mall" />
                < property name=”state” value="Karnataka" />
                < property name=”zipcode” value="560076" />

</bean>
```

Injecting Null and Empty String values

```
<bean id="customer" class="com.kalibermind.domain.Customer">
        <property name="empName" value="" /> <!-- Empty String Value -->

</bean>
```

The preceding example is equivalent to the following Java code:
customer.setEmpName(“”)

The <null /> element handles null values. For example:

```
<bean id="customer" class="com.kalibermind.domain.Customer">
<property name="email">
        <null /> <!-- Null Value -->
</property>
```

The above configuration is equivalent to the following Java code:
customer.setEmail(null)

Injecting Collections(List, Set, Map) and Properties

In the **<list />, <set />, <map />,** and **<props />** elements, you set the properties and arguments of the Java Collection types **List, Set, Map,** and **Properties**, respectively.

All these elements are sub elements of **<property>** and **<constructor-arg>**

```
package com.kalibermind.domain;

import java.util.*;

public class Employee {
        private long id;
        private String firstName;
        private String lastName;
        private String email;
        private List<Long> mobiles;
        private Set<String> projectSet;
        private Map<String, String> projects;
        private Properties productDesc;
        public Employee() {
                System.out.println("Employee object is created");
        }
        //Getters and Setters
}
```

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd">

        <bean id="employee" class="com.kalibermind.domain.Employee">
            <property name="id" value="1001" />
            <property name="firstName" value="Chandra" />
            <property name="lastName" value="Verma" />
            <property name="email" value="CPVerma@gmail.com" />
            <property name="mobiles">
                <list>
                    <value>8123417647</value>
                    <value>8123417777</value>
                    <value>9899965342</value>
                    <value>7676543233</value>
                </list>
            </property>

            <property name="projectSet">
                <set>
                    <value>Hotels</value>
                    <value>Ecommerce</value>
                    <value>Health Care</value>
                    <value>null</value>
                </set>
            </property>

            <property name="projects">
            <map>
                <entry key="ecommerce" value="FreakCoders eCommerce" />
                <entry key="healthcare" value="FreakCoders Healthcare" />
                <entry key="travels" value="FreakCoders Hotels" />
            </map>
           </property>

            <property name=”productDesc”>
            <props>
                <prop key="color">Red</prop>
                <prop key="ram">4GB</prop>
                <prop key="memory">64GB</prop>
                <prop key="screen">5</prop>
            </props>
            </property>

         </bean>
</beans>
```

Note: The value of a map key or value, or a set value, can also again be any of the following elements: **<bean>,<ref>,<list>,<set>,<map>,<props>,<value>,<null>**

Injecting References of other beans (Collaborators or Dependent Objects)

The references bean is a dependency of the bean whose property will be set, and it is initialized on demend as needed before the property is set.

To inject one bean object as dependency of another bean, the <constructor-arg> and <property> elements of **<bean>** defines ref attribute and **<ref>** element. By using these elements you can inject any reference object.

```
package com.Biditvats.domain;

public class Customer {
	private long id;
	private String firstName;
	private String lastName;
	private String email;
	private long mobile;
	private Address address;

	public Customer() { }
	//Getters and Setters
}
```

```
package com.Biditvats.domain;

public class Address {
	private long id;
	private String houseNo;
	private String street;
	private String city;
	private String landmark;
	private String state;
	private long zipcode;

	public Address() { }
	//Getters and Setters
}
```

Injecting Dependency Using Constructor

```
public Customer(long id, String firstName, String lastName, String email, long mobile, Address address) {
		super();
		this.id = id;
		this.firstName = firstName;
		this.lastName = lastName;
		this.email = email;
		this.mobile = mobile;
		this.address = address;
	}
```

```
<bean id="customer" class="com.Biditvats.domain.Customer">
				<constructor-arg value="1101" />
				<constructor-arg value="Bidit" />
				<constructor-arg value="Vats" />
				<constructor-arg value="Biditvats@gmail.com" />
```

```
                    <constructor-arg value="9916712669" />
                    <constructor-arg ref="address" />
</bean>


<bean id="address" class="com.Biditvats.domain.Address">
                <property name="id" value="1101" />
                <property name="houseNo" value="H-1318, Ground floor" />
                <property name="street" value="JP Nagar, 2nd Phase" />
                <property name="city" value="Bengaluru" />
                <property name="landmark" value="Next to Bangalore Central Mall" />
                <property name="state" value="Karnataka" />
                <property name="zipcode" value="560076" />

</bean>

<constructor-arg>
        <ref bean =”address”/>
</constructor-arg>
```

Injecting Dependency Using Setter

```
<bean id="customer" class="com.Biditvats.domain.Customer">
                <property name="id" value="1001" />
                <property name="firstName" value="Bidit" />
                <property name="lastName" value="Jha" />
                <property name="email" value="Biditvats@gmail.com" />
                <property name="mobile" value="9916712669" />
                <property name="address" ref="address" />
</bean>

<bean id="address" class="com.Biditvats.domain.Address">
                <property name="id" value="1101" />
                <property name="houseNo" value="H-1318, Ground floor" />
                <property name="street" value="JP Nagar, 2nd Phase" />
                <property name="city" value="Bengaluru" />
                <property name="landmark" value="Next to Bangalore Central Mall" />
                <property name="state" value="Karnataka" />
                <property name="zipcode" value="560076" />

</bean>

<property>
        <ref bean =”address”/>
</property>
```

Autowiring Collaborators

Creating a bean object and injecting that object into dependent bean object without passing the reference of that object is called **autowiring.** This complete process is handled by IoC container.
The Spring IoC container can **autowire** relationships between collaborating beans. You can allow Spring to resolve collaborators automatically for your bean by inspecting the contents of the **ApplicationContext.**

Autowiring Modes

Mode	Description
no	(Default) No autowiring. Bean references must be defined via a ref element.
byName	Autowiring by property name, property name in collaborator bean and the id of dependent object must be same. and the setter method is required.
byType	Autowiring by type, property name in collaborator bean and the id of dependent object may be same or different same. setter method is required. If there are two same type objects, occurs ambiguity.
constructor	It uses the Autowiring by type, but parameterized constructor is required.
autodetect	Actually spring autowire=”autodetect” frist will works as Spring Autowiring constructor if not then works as Spring Autowiring byType, byType means setter injection.

NOTE:
When using XML-based configuration metadata, you specify autowire attribute of the <bean />element. The **autowire** attribute has four modes. You specify autowiring per bean and thus can choose which ones to *autowire*.

Advantages of Autowiring:

1. Autowiring reduces the need to specify properties or constructor arguments.
2. Autowiring can update a configuration as your objects evolve.

Disadvantages of Autowiring:

1. You *cannot* autowire primitives, Strings, and Classes (and arrays of such simple properties).
2. Explicit dependencies in property and constructor-arg settings always override autowiring.
3. Spring is careful to avoid guessing in case of ambiguity that might have unexpected results.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd">

        <!-- Injecting the bean without passing ref is called Autowiring.
            byName autowiring required setter method because it uses setter DI.
            In byName autowiring, property name and bean ID must be same -->

        <bean id="customer" class="com.Biditvats.domain.Customer" autowire="byName">
                <property name="id" value="1001" />
                <property name="firstName" value="Bidit" />
                <property name="lastName" value="Jha" />
                <property name="email" value="Biditvats@gmail.com" />
                <property name="mobile" value="9916712669" />
        </bean>

        <bean id="address" class="com.Biditvats.domain.Address">
                <property name="id" value="1101" />
```

```
        <property name="houseNo" value="H-1318, Ground floor" />
        <property name="street" value="JP Nagar, 2nd Phase" />
        <property name="city" value="Bengaluru" />
        <property name="landmark" value="Next to Bangalore Central Mall" />
        <property name="state" value="Karnataka" />
        <property name="zipcode" value="560076" />

    </bean>

</beans>
```

Defining Inner Bean

1. Defining **<bean />** element inside the **<property />** or **<constructor-arg />** elements is called inner bean.
2. An inner bean definition *does not require* a defined id or name, if specified, the container does not use that value as an identifier.
3. The container also **ignores** the **scope** of inner bean.
4. Inner beans are always created with the outer bean.
5. It is not possible to inject inner beans into collaborating beans other than into the enclosing.

```
<bean id="customer" class="com.Biditvats.domain.Customer" >
        <property name="id" value="1001" />
        <property name="firstName" value="Bidit" />
        <property name="lastName" value="Jha" />
        <property name="email" value="Biditvats@gmail.com" />
        <property name="mobile" value="9916712669" />
        <property name="address">
            <bean id="address" class="com.Biditvats.domain.Address">
                <property name="id" value="1101" />
                <property name="houseNo" value="H-1318, Ground floor" />
                <property name="street" value="JP Nagar, 2nd Phase" />
                <property name="city" value="Bengaluru" />
                <property name="landmark" value="Next to Bangalore Central Mall" />
                <property name="state" value="Karnataka" />
                <property name="zipcode" value="560076" />

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

Defining Compound Property Names

You can use **compound** or **nested property** names when you set bean properties, as long as all components of the path except the final property name are not null.

```
<bean id="customer" class="com.Biditvats.domain.Customer" >
        <property name="id" value="1001" />
        <property name="firstName" value="Bidit" />
        <property name="lastName" value="Jha" />
        <property name="email" value="Biditvats@gmail.com" />
        <property name="mobile" value="9916712669" />
        <property name="address" ref="address" />

            <!-- Compound properties -->
            <property name="address.id" value="1101" />
```

```
<property name="address.houseNo" value="H-1318, Ground floor" />
<property name="address.street" value="JP Nagar, 2nd Phase" />
<property name="address.city" value="Bengaluru" />
<property name="address.landmark" value="Next to Bangalore Central Mall" />
<property name="address.state" value="Karnataka" />
<property name="address.zipcode" value="560076" />
```

</bean>

<bean id="address" class="com.Biditvats.domain.Address">
</bean>

Lazy-Initialized Beans

1. By default, **ApplicationContext** uses **eagerly** create and configure all singleton beans as part of the initialization process. ***Default lazy-init="false"***
2. you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized.
3. A lazy-initialized bean tells the IoC container to create a bean instance when it is ***first requested,*** rather than at ***startup.***
4. In XML, this behavior is controlled by the **lazy-init** attribute on the **<bean />** element.

<bean id=”address” class=**”com.Biditvats.domain.Address” lazy-init=”true”** />

You can also control lazy-initialization at the container level by using the **default-lazy-init** attribute on the **<beans/>** element; for example:

```
<beans default-lzy-init=”true”>
    <! -- no beans will be pre-instantiated…  -- >
</beans>
```

Bean scopes

1. You can also define the scope of the objects created from a particular bean definition.
2. This approach is powerful and flexible in that you can choose the scope of the objects you create through configuration.
3. The Spring Framework supports seven scopes, five of which are available only if you use a web-aware ApplicationContext.

Scope	Description
singleton	Singleton is a default Scope of Spring Bean. It creates a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext .

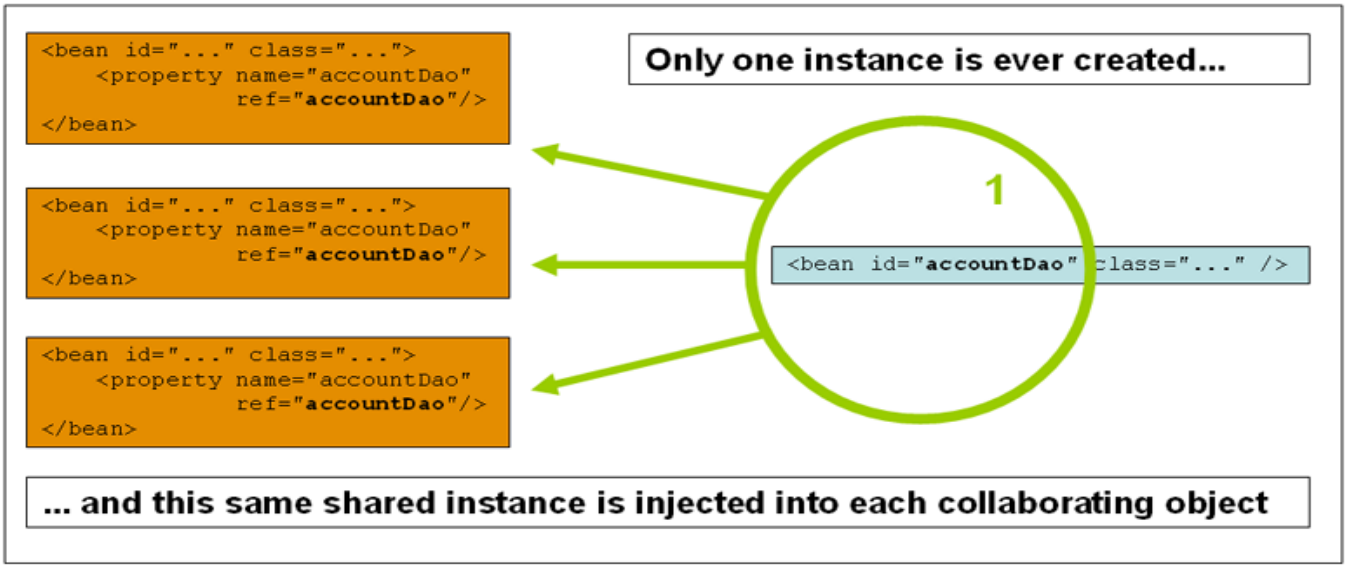
session	Scopes a single bean definition to the lifecycle of an HTTP Session . Only valid in the context of a web-aware Spring ApplicationContext .
application	Scopes a single bean definition to the lifecycle of a ServletContext . Only valid in the context of a web-aware Spring ApplicationContext .
globalSession	Scopes a single bean definition to the lifecycle of a global HTTP Session . Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext .
websocket	Scopes a single bean definition to the lifecycle of a WebSocket . Only valid in the context of a web-aware Spring ApplicationContext .

NOTE:
By using scope attribute of <bean> element, you can define the scope of the bean, passing value from above table. The default scope of bean is singleton(per IoC one Object only).

<bean id=”customer” class=”com.Biditvats.domain.Customer” **scope=”singleton”** />

Singleton Scope

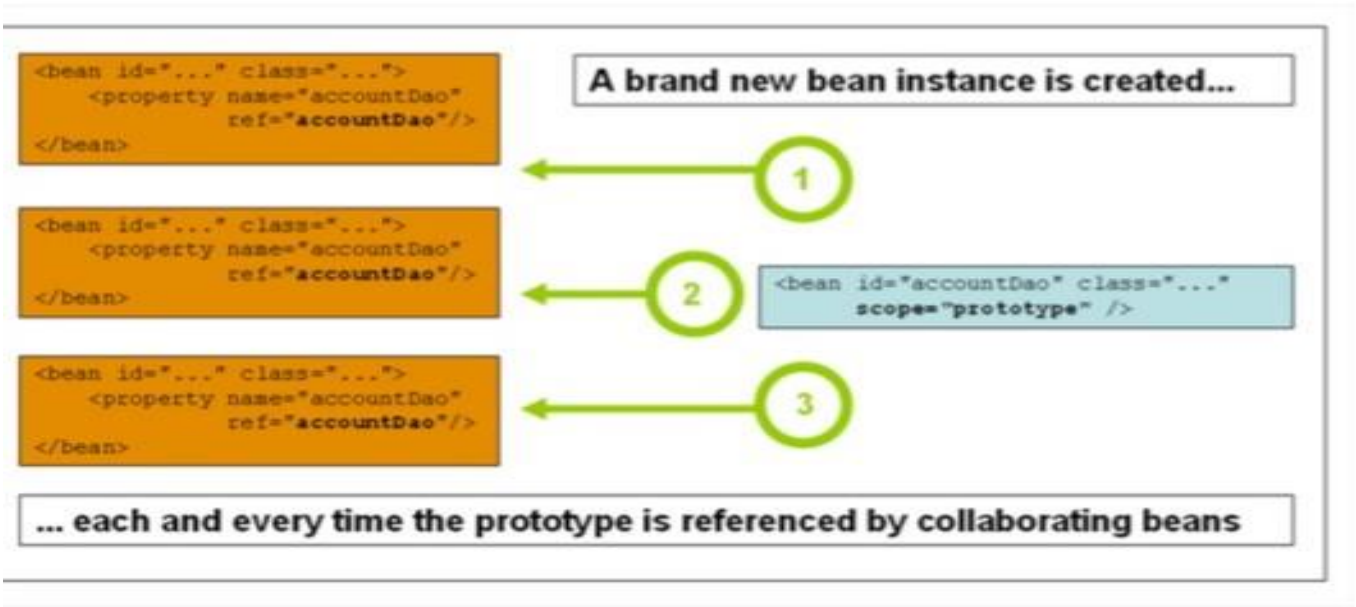
1. The Spring IoC container creates *exactly one instance* of the object defined by that bean definition.
2. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.



<bean id=”customer” class=”com.Biditvats.domain.Customer” **scope=”singleton”** />

Prototype Scope

1. The prototype scope of bean deployment results in the creation of a *new bean* instance every time a request for that specific bean is made.
2. That is, the bean is injected into another bean or you request it through a getBean() method call on the container.
3. As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.



<bean id="address" class="com.Biditvats.domain.Address" **scope="prototype"** />

Singleton with Prototype Bean Dependencies

1. When you use **singleton-scoped** beans with dependencies on **prototype beans**, be aware that dependencies are resolved at instantiation time.
2. Thus if you dependency-inject a **prototype-scoped** bean into a **singleton-scoped** bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean.

However, suppose you want the **singleton-scoped** bean to acquire a new instance of the **prototype-scoped** bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies.

Request, session, global session, application, and WebSocket scopes

The **request, session, globalSession, application,** and **websocket** scopes are only available if you use a **web-aware Spring ApplicationContext** implementation (such as **XmlWebApplicationContext**).

If you use these scopes with regular Spring IoC containers such as the **ClassPathXmlApplicationContext,** an **IllegalStateException** will be thrown complaining about an unknown bean scope.

Lifecycle of Spring Bean

To interact with the container's management of the bean lifecycle, you can implement the lifecycle callbacks. There are two lifecycle callbacks-

Initialization Callbacks

Allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. Spring supports three ways to define Initialization Callbacks-

1. By implementing **org.springframework.beans.factory.InitializingBean** interface in bean class that contains only one method **void afterPropertiesSet() throws Exception;,** all initialization work you can define in this method.

2. By **init-method** attribute of **<bean>** element. you can define any method name, here **init** is a method name. With ***Java config***, you use the **initMethod** attribute of **@Bean.**
3. Using **JSR-250** annotation **@PostConstruct**
   @PostConstruct
   Public void init( ) {
   //Initialization Code
   }

Destruction Callbacks

Allows a bean to get a callback when the container containing it is destroyed. Spring supports three ways to define Destruction Callbacks-

1. By implementing **org.springframework.beans.factory.DisposableBean** interface in bean class that contains only one method **void destroy()** throws Exception;.
2. By **destroy-method** attribute of **<bean>** element. you can define any method name, here **destroy** is a method name. With ***Java config,*** you use the **destroyMethod** attribute of **@Bean**.
3. Using JSR-250 annotation **@PreDestroy**
   @PreDestroy
   Public void destroy( ) {
   // Destruction Code
   }

Default Initialization And Destroy Methods

1. you can also define default initialization and destroy methods for all beans.
2. you use **default-init-method** and **default-destroy-method** attribute of **<beans>** element.

<beans default-init-method=”init” default-destroy-method=”destroy”>
<!-- Define bean -->
</beans>

Spring Annotation Based Configuration

An alternative to XML setups is provided by annotation-based configuration, Spring version-2.5 added this feature which allow spring-based-configuration for spring applications. Instead of using XML to describe a bean wiring, the developer can use annotations on the relevant class, method, or field declaration.

By default all the annotations are disabled in spring framework, to use annotation-based-configuration, first you need to enable annotations. To enable annotation, add context namespace in spring configuration XML file

Commonly Used Annotations

Spring Annotation	JSR-330 Annotation	Description
`@Component`	`@Named/@ManagedBean`	Defines the bean class
`@Autowired`	`@Inject`	Defines the autowiring
`@Scope("singleton")`	`@Singleton`	Defines the scope of bean

`@Qualifier`	`@Qualifier / @Named`	Defines the name of the bean
`@Lazy`	---	To achieve lazy-initialization
`@Required`	---	To set required property (only with setter)
`@Value`	---	To set property value by setter

Application

spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xmlns:context="http://www.springframework.org/schema/context"
		xsi:schemaLocation="http://www.springframework.org/schema/beans
			http://www.springframework.org/schema/beans/spring-beans.xsd
			http://www.springframework.org/schema/context
			http://www.springframework.org/schema/context/spring-context.xsd">

	<context:annotation-config />
	<context:component-scan base-package="com.Biditvats.domain" />

</beans>
```

Address.java

```
package com.Biditvats.domain;

import org.springframework.beans.factory.annotation.Required;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class Address {
	private long id;
	private String houseNo;
	private String street;
	private String city;
	private String landmark;
	private String state;
	private long zipcode;

	public Address() {
```

```
        System.out.println("Address Object is created");
}

public long getId() {
        return id;
}

@Value("1100")
public void setId(long id) {
        this.id = id;
}

public String getHouseNo() {
        return houseNo;
}

@Value("H-1318, Ground floor")
public void setHouseNo(String houseNo) {
        this.houseNo = houseNo;
}

public String getStreet() {
        return street;
}

@Value("JP Nagar, 2nd Phase")
public void setStreet(String street) {
        this.street = street;
}

public String getCity() {
        return city;
}

@Required
@Value("Bangalore")
public void setCity(String city) {
        this.city = city;
}

public String getLandmark() {
        return landmark;
}

@Value("Next to Bangalore Central Mall")
public void setLandmark(String landmark) {
        this.landmark = landmark;
}

public String getState() {
        return state;
}

@Value("Karnataka")
public void setState(String state) {
```

```
		this.state = state;
	}

	public long getZipcode() {
		return zipcode;
	}

	@Value("560076")
	public void setZipcode(long zipcode) {
		this.zipcode = zipcode;
	}

	@Override
	public String toString() {
		return "Address ID:" +id
				+"\nHouse No: " +houseNo
				+"\nStreet: " +street
				+"\nCity: " +city
				+"\nLandmark: " +landmark
				+"\nState: " +state
				+"\nZipcode: " +zipcode;
	}
}
```

Employee.java

```
package com.Biditvats.domain;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Required;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@Component
@Qualifier("employee")
@Lazy(true)
public class Employee {
	private long id;
	private String name;
	private String email;
	private String dept;
	private double salary;

	@Autowired
	private Address address;

	public Employee() {
		System.out.println("Employee object is created");
	}
```

```
public long getId() {
	return id;
}

@Required
@Value("1001")
public void setId(long id) {
	System.out.println("setId() Method is called");
	this.id = id;
}

public String getName() {
	return name;
}

@Value("Bidit vats")
public void setName(String name) {
	this.name = name;
}

public String getEmail() {
	return email;
}

@Value("Biditvats@gmail.com")
public void setEmail(String email) {
	this.email = email;
}

public String getDept() {
	return dept;
}

public void setDept(String dept) {
	this.dept = dept;
}

public double getSalary() {
	return salary;
}

public void setSalary(double salary) {
	this.salary = salary;
}



public Address getAddress() {
	return address;
}

public void setAddress(Address address) {
	this.address = address;
}
```

```
	@PostConstruct
	public void init() {
		System.out.println("init() - Employee Bean is Initialized!");
	}

	@PreDestroy
	public void destroy() {
		System.out.println("destroy() - Employee Bean is destroyed!");
	}

}
```

TestEmployee.java

```
package com.Biditvats.test;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.Biditvats.domain.Address;
import com.Biditvats.domain.Employee;

public class TestEmployee {

	public static void main(String[] args) {
		AbstractApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

		Employee employee1 = context.getBean(Employee.class);

		Address address = context.getBean(Address.class);

		Employee employee2 = context.getBean(Employee.class);

		System.out.println("Employee ID:" +employee1.getId());
		System.out.println("Empoyee Name:" +employee1.getName());
		System.out.println("Employee email:" +employee1.getEmail());
		System.out.println(employee1.getAddress());
	context.registerShutdownHook();
	}
}
```

Java-based container configuration

Address.java

```
package com.Biditvats.spring.bean;

public class Address {
	private long addressId;
	private String houseNo;
	private String street;
	private String city;
	private String landmark;
	private String state;
	private long zipcode;

	public Address() {
```

```
		System.out.println("Address Object is created");
	}

	public Address(long addressId, String houseNo, String street, String city, String landmark, String
state,long zipcode) {
		super();
		this.addressId = addressId;
		this.houseNo = houseNo;
		this.street = street;
		this.city = city;
		this.landmark = landmark;
		this.state = state;
		this.zipcode = zipcode;
	}
	/ / Corresponding Getters and Setters
}
```

Employee.java

```
package com.Biditvats.spring.bean;

public class Employee {
	private long empId;
	private String empName;
	private String email;
	private String dept;
	private double salary;
	private Address address;

	public Employee() {
		System.out.println("Employee object is created");
	}

	public Employee(long empId, String empName, String email, String dept, double salary,Address
address) {
		super();
		this.empId = empId;
		this.empName = empName;
		this.email = email;
		this.dept = dept;
		this.salary = salary;
		this.address = address;
	}
	/ / Corresponding Getters and Setters
}
```

JavaConfig.java

```
package com.Biditvats.spring.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.Biditvats.spring.bean.Address;
import com.Biditvats.spring.bean.Employee;

@Configuration
```

```
public class JavaConfig {

	@Bean
	public Employee getEmployee() {
		return new Employee();
	}

	@Bean
	public Address getAddress() {
		return new Address();
	}

}
```

TestEmployee.java

```
package com.Biditvats.spring.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.Biditvats.spring.bean.Employee;
import com.Biditvats.spring.config.JavaConfig;

public class TestEmployee {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		ApplicationContext context = new AnnotationConfigApplicationContext(JavaConfig.class);

		Employee emp = context.getBean(Employee.class);

		System.out.println("Employee name:" +emp.getEmpName());
	}

}
```