SPRING FRAMEWORK

[Spring Framework:](#_sltjkhxdk5bl)

[Spring Framework architecture:](#_m1pkvavf48yn)

[Inversion Of Control and Dependency Injection :](#_8qoddoulo3cu)

[IOC containers:](#_39fat0821ctg)

[Spring BeanFactory Container:](#_r0zpxsy90jch)

[Spring ApplicationContext Container:](#_oyix6g5lu45u)

[Spring Beans:](#_k439bo6f5bvt)

[Bean LifeCycle:](#_o9625ujnzycu)

[Dependency Injection:](#_wu2wu0ygnhba)

[Constructor-based DI](#_orrj21lcxgsr)

[Setter-based DI](#_afrr7a3h0ab1)

[Exercise:](#_wdf41wug6xn8)

[Steps:](#_353w90cdi2g5)

[Source code link:](#_785mysy1w7l3)

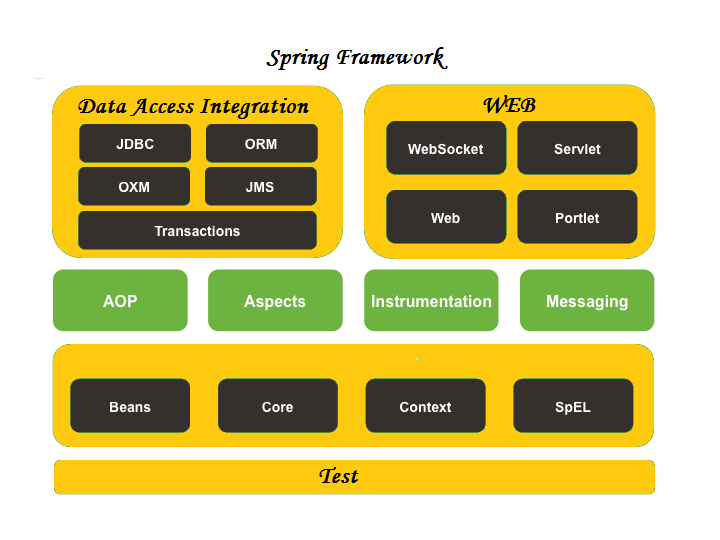
# Spring Framework:

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

## 

## Spring Framework architecture:



## Inversion Of Control and Dependency Injection :

These are the design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain.

1. **class** Employee{
2. Address address;
3. Employee(){
4. address=**new** Address();
5. }
6. }

In the above code,dependency between employee and address is high or it is tightly coupled .This coupling can be minimised with the help of IOC . Inversion of control makes the code loosely coupled.

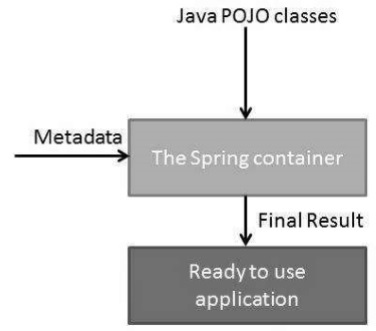
In Spring framework, IOC container is responsible to inject the dependency.

### IOC containers:

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI(Dependency Injection) to manage the components that make up an application.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides the following two distinct types of containers:

Spring BeanFactory Container

Spring ApplicationContext Container

#### Spring BeanFactory Container:

This is the simplest container providing the basic support for DI and defined by the **org.springframework.beans.factory.BeanFactory** interface.

The most commonly used BeanFactory implementation is the **XmlBeanFactory** class.

This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

#### Spring ApplicationContext Container:

The Application Context is Spring's advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. Additionally, it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by ***org.springframework.context.ApplicationContext*** interface.

The most commonly used ApplicationContext implementations are − (First two implementations are used for loading the definitions of the beans from an XML file.)

1. **FileSystemXmlApplicationContext -** need to provide the full path of the XML bean configuration file to the constructor.

Example: ApplicationContext context = new FileSystemXmlApplicationContext  
 ("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");

1. **ClassPathXmlApplicationContext -** do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.

Example: ApplicationContext context = new ClassPathXmlApplicationContext ("Beans.xml");

1. **WebXmlApplicationContext -** This container loads the XML file with definitions of all beans from within a **web application.**

XmlWebApplicationContext used to read beans.xml spring application context configuration from web location.

* setConfigLocation method used to set path of the bean configuration file.
* setServletContext method used to set servletcontext instance.
* refresh method used to refresh the bean memory based on new config file.

XmlWebApplicationContext context = new XmlWebApplicationContext();

context.setConfigLocation("/WEB-INF/beans.xml");

context.setServletContext(request.getServletContext());

context.refresh();

### Spring Beans:

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container.

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

class

This attribute is mandatory and specifies the bean class to be used to create the bean.

name

This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).

Example:

<!-- A simple bean definition -->  
 <bean id = "..." class = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>

lazy-initialization mode

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.

Example:

<!-- A bean definition with lazy init set on -->  
 <bean id = "..." class = "..." lazy-init = "true">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>

#### Bean LifeCycle:

When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required. To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately **upon instantiation**. Similarly, destroymethod specifies a method that is called **just before a bean is removed** from the container.

initialization method

A callback to be called just after all necessary properties on the bean have been set by the container.

Example:

<!-- A bean definition with initialization method -->  
 <bean id = "..." class = "..." init-method = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>

destruction method

A callback to be used when the container containing the bean is destroyed.

Example:

<!-- A bean definition with destruction method -->  
 <bean id = "..." class = "..." destroy-method = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>

Destroying Context object:

((ClassPathXmlApplicationContext) context).close();

Or

context.registerShutdownHook();

The above two call will call the “destroy method” specified in destroy-method inside bean.xml.

### Dependency Injection:

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider an example similar to the employee address example stated previously.

public class TextEditor {  
 private SpellChecker spellChecker;  
   
 public TextEditor() {  
 spellChecker = new SpellChecker();  
 }  
}

As in the previous example dependency between TextEditor and SpellChecker is high. This is solved by the Dependency Injection.

public class TextEditor {  
 private SpellChecker spellChecker;  
   
 public TextEditor(SpellChecker spellChecker) {  
 this.spellChecker = spellChecker;  
 }  
}

By inversion of control, the dependency between TextEditor and SpellChecker is minimised.

We have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor.** Thus the flow of control has been "inverted" by **Dependency Injection (DI)** because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

DI exists in two major variants:

#### Constructor-based DI

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

if you are passing a value directly then you should use **value** attribute

<beans>  
 <bean id = "exampleBean" class = "examples.ExampleBean">  
 <constructor-arg type = "int" value = "2001"/>  
 <constructor-arg type = "java.lang.String" value = "Zara"/>  
 </bean>  
</beans>

Another way to pass values to constructor is:

<beans>  
  
 <bean id = "exampleBean" class = "examples.ExampleBean">  
 <constructor-arg index = "0" value = "2001"/>  
 <constructor-arg index = "1" value = "Zara"/>  
 </bean>  
  
</beans>

in case you are passing a reference to an object, you need to use **ref** attribute of <constructor-arg> tag

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">  
 <constructor-arg ref = "spellChecker"/>  
 </bean>

#### Setter-based DI

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">  
 <property name = "spellChecker" ref = "spellChecker"/>  
 </bean>  
  
 <bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

## Exercise:

Implement dependency injection and IoC in a basic Spring application and explore the bean lifecycle ( Create a Maven project )

### Steps:

* Create a Maven project
* You will be getting a pom.xml file by default on creation of a maven project.
* Add spring dependencies in this pom.xml files , all these dependendencies will be downloaded and stored under maven dependencies.
* Now create a simple POJO class that has name and age attributes,generate getters and setters, have a method that prints both name and age.
* Have initmethod and cleanup methods that gets invoked when bean is created and destroyed.
* Now create applicationContext.xml file inside source/main/resources folder.
* Define the bean class inside applicationContext.xml file.You can set values to the class with the help of setter methods.

<bean id="helloWorld" class="com.zilker.bean.HelloWorld" init-method="initIt" destroy-method="cleanUp">

<property name="name" value="anu" />

<property name="age" value="16" />

</bean>

* Now this bean class(HelloWorld) can be referenced with the help of this id “helloWorld”.
* Create another java class(HelloWorldOperation) inside a different package and use one of IOC implementation variants to access the bean from XML file and create an object of the bean type, this object can be further used to access the methods inside the bean class.

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

HelloWorld service = (HelloWorld) context.getBean("helloWorld") ; //creates bean object and initIt method gets called//

String message = service.sayHello();

System.out.println(message);

((ClassPathXmlApplicationContext) context).close(); //used to close the bean definition -- and cleanUp method gets invoked//

* Now for constructor-based DI:
* Create another class under bean package with same attributes with constructors and init and cleanUp method as before.
* Now in applicationContext.xml,
* <bean id="helloWorld1" class="com.zilker.bean.ConstructorDependency" init-method="initIt" destroy-method="cleanUp">

<constructor-arg value="sruthi"></constructor-arg> <constructor-arg value="20" type="int"></constructor-arg>

</bean>

* Perform the same steps as done before in HelloworldOperation except for referencing the new bean id(helloWorld1) instead of helloworld.

### Source code link:

[https://github.com/sruthiviswanathan/Zterns-19-Sruthi/tree/master/SpringFramework](https://github.com/sruthiviswanathan/ServletTasks/tree/master/helloworld)