Spring Frameworks: Introduction to Spring Framework,POJO Programming Model, Lightweight Containers(Spring IOC container, Configuration MetaData, Configuring and using the Container) Dependency Injection with Spring- Setter Injection, Constructor Injection, Circular Dependency, Overriding Bean, Auto Wiring Bean Looksup, Spring Manage Beans)

**What is the difference between Java and Spring?**

The below table represents the differences between Java and Spring:

|  |  |
| --- | --- |
| **Java** | **Spring** |
| Java is one of the prominent programming languages in the market. | Spring is a Java-based open-source application framework. |
| Java provides a full-highlighted Enterprise Application Framework stack called Java EE for web application development | Spring Framework comes with various modules like Spring MVC, Spring Boot, Spring Security which provides various ready to use features for web application development. |
| Java EE is built upon a 3-D Architectural Framework which are Logical Tiers, Client Tiers and Presentation Tiers. | Spring is based on a layered architecture that consists of various modules that are built on top of its core container. |

**Dependency Injection**

When class A uses some functionality of class B, then its said that class A has a dependency of class B.

In Java, before we can use methods of other classes, we first need to create the object of that class (i.e. class A needs to create an instance of class B).

So, transferring the task of creating the object to someone else and directly using the dependency is called dependency injection.

Let’s say we have a car class which contains various objects such as wheels, engine, etc.

Here the car class is responsible for creating all the dependency objects. Now, what if we decide to ditch **MRFWheels**in the future and want to use **Yokohama** Wheels?

We will need to recreate the car object with a new Yokohama dependency. But when using dependency injection (DI), we can change the Wheels at runtime (because dependencies can be injected at runtime rather than at compile time).

You can think of DI as the middleman in our code who does all the work of creating the preferred wheels object and providing it to the Car class.

It makes our Car class independent from creating the objects of Wheels, Battery, etc.

#### There are basically three types of dependency injection:

1. **constructor injection:** the dependencies are provided through a class constructor.
2. **setter injection:** the client exposes a setter method that the injector uses to inject the dependency.
3. **interface injection:** the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a [setter method](https://en.wikipedia.org/wiki/Setter_method) that accepts the dependency.

**So now its the dependency injection’s responsibility to:**

1. Create the objects
2. Know which classes require those objects
3. And provide them all those objects

If there is any change in objects, then DI looks into it and it should not concern the class using those objects. This way if the objects change in the future, then its DI’s responsibility to provide the appropriate objects to the class. Inversion of control —the concept behind DI -This states that a class should not configure its dependencies statically but should be configured by

A class should concentrate on fulfilling its responsibilities and not on creating objects that it requires to fulfill those responsibilities. And that’s where **dependency injection** comes into play: it provides the class with the required objects.

#### Benefits of using DI

1. Helps in Unit testing.
2. Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
3. Extending the application becomes easier.
4. Helps to enable loose coupling, which is important in application programming.

#### Disadvantages of DI

1. It’s a bit complex to learn, and if overused can lead to management issues and other problems.
2. Many compile time errors are pushed to run-time.
3. Dependency injection frameworks are implemented with reflection or dynamic programming. This can hinder use of IDE automation, such as “find references”, “show call hierarchy” and safe refactoring.

**Inversion Of Control**

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.

The advantages of this architecture are:

* decoupling the execution of a task from its implementation
* making it easier to switch between different implementations
* greater modularity of a program
* greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

**Spring IoC Container**

Spring IoC stands for Inversion of Control. It is the heart of the Spring Framework. The important tasks performed by the IoC container are:

* Instantiating the bean
* Wiring the beans together
* Configuring the beans
* Managing the bean’s entire life-cycle

The IoC container receives metadata from either an XML file, Java annotations, or Java code and works accordingly. IoC adds the flexibility and control of application, and provides a central place of configuration management for Plain Old Java Objects (POJO) of our application. This diagram represents an abstract view of the working of Spring Framework. It shows how Spring makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

There are two types of IoC containers:

1. BeanFactory
2. ApplicationContext

Lets discuss them in detail.

**BeanFactory**

* It is an interface defined in org.springframework.beans.factory.**BeanFactory**.
* Bean Factory provides the basic support for Dependency Injection.
* It is based on factory design pattern which creates the beans of any type.
* BeanFactory follows lazy-initialization technique which means beans are loaded as soon as bean factory instance is created but the beans are created only when *getBean()* method is called.
* The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, you need to create the instance of XmlBeanFactory class as shown below:
* BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("beans.xml"));

**ApplicationContext**

* It is an interface defined in org.springframework.context.**ApplicationContext**.
* It is the advanced Spring container and is built on top of the BeanFactory interface.
* ApplicationContext supports the features supported by Bean Factory but also provides some additional functionalities.
* ApplicationContext follows eager-initialization technique which means instance of beans are created as soon as you create the instance of Application context.
* The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. You need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as shown below:

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

Diagram

Description automatically generated

**Configuration Metadata:**

as the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format.

In addition to XML

Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.

Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework.

Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the @Configuration, @Bean, @Import and @DependsOn annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage.

XML-based configuration metadata shows these beans configured as <bean/> elements inside a top-level <beans/> element.

The following example shows the basic structure of XML-based configuration metadata:

<?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

https://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<bean id="..." class="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions go here -->

</beans>

The id attribute is a string that identifies the individual bean definition.

The class attribute defines the type of the bean and uses the fully qualified classname.

The value of the id attribute refers to collaborating objects.

#### **Instantiating a Container**

The location path or paths supplied to an ApplicationContext constructor are resource strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java CLASSPATH, and so on.

Syntax:

private static ApplicationContext appCon;

appCon=new ClassPathXmlApplicationContext("Context.xml");

**Beans:**

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML <bean/> definitions).

Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:

* A package-qualified class name: typically, the actual implementation class of the bean being defined.
* Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
* References to other beans that are needed for the bean to do its work. These references are also called collaborators or dependencies.
* Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.

This metadata translates to a set of properties that make up each bean definition. The following table describes these properties:

**name / id:**

This attribute specifies the bean unique identifier. In XML based configuration metadata, you use the id and/or name attributes to specify the bean identifier.

**class:**

This attribute is mandatory and specify the bean class to be used to create the bean. You should specify fully qualified class name. Include package name.

**scope:**

This attribute specifies the scope of the objects created from a particular bean definition. The scope values can be prototype, singleton, request, session, and global session.

Prototype scope indicates multiple objects can be created

**constructor-arg:**

This is used to inject the dependencies through bean constructor.

**properties:**

This attribute is used to inject the dependencies through setter method.

**autowiring mode:**

This is used to inject the dependencies.

**lazy-init (lazy-initialization mode):**

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

**init-method (initialization method):**

A callback to be called just after all necessary properties on the bean have been set by the container. This is part of bean lifecycle.

**destroy-method (destruction method):**

A callback to be used when the container containing the bean is destroyed. This is part of bean lifecycle.

**Bean Scopes**

|  |  |
| --- | --- |
| **Scope** | **Description** |
| [singleton](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes-singleton) | (Default) Scopes a single bean definition to a single object instance for each Spring IoC container. |
| [prototype](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes-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](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes-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](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes-application) | Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext. |

#### **Constructor-based dependency injection**

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

The <constructor-arg> subelement of <bean> is used for constructor injection. e.g:

<constructor-arg value="101" type="int"></constructor-arg>

By default when the Spring container loads the bean, it instantiates the bean with the default constructor.

But you can also define a constructor argument in bean definition, using an argument constructor.

Constructor argument resolution matching occurs using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated.

Use the index attribute to specify explicitly the index of constructor arguments. For example:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

As of Spring 3.0 you can also use the constructor parameter name for value disambiguation:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateanswer" value="42"/>

</bean>

**Example:**

**Model:**

package edu.met;

public class FullName {

private String Fname;

private String Mname;

private String Lname;

public FullName(String fname, String mname, String lname) {

super();

Fname = fname;

Mname = mname;

Lname = lname;

}

public FullName() {

super();

}

public String toString()

{

return Fname+" "+Mname+" "+Lname;

}

}

package edu.met;

public class Employee {

private int eid;

private FullName ename;

public Employee(int eid, FullName ename) {

super();

this.eid = eid;

this.ename = ename;

}

void displayInfo()

{

System.out.println("Id:"+eid);

System.out.println("Ename:"+ename);

}

}

**Bean Definition File:**

<?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="FN" class="edu.met.FullName">

<constructor-arg value="Omprakash"></constructor-arg>

<constructor-arg value="Laxman"></constructor-arg>

<constructor-arg value="Mandge"></constructor-arg>

</bean>

<bean id="EM" class="edu.met.Employee">

<constructor-arg value="1"></constructor-arg>

<constructor-arg>

<ref bean="FN"/>

</constructor-arg>

</bean>

</beans>

Setter-based dependency injection

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.

The <property> subelement of <bean> is used for setter injection.e.g:

<property name="id" value="101"></property>

Setter-based Dependency Injection is accomplished by the container calling setter methods on your beans after invoking a no-argument.

**Dependency Resolution Process**

The container performs bean dependency resolution as follows:

1. The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code or annotations.
2. For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, *when the bean is actually created*.
3. Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
4. 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. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created, including the validation of whether bean reference properties refer to valid beans. However, the bean properties themselves are not set until the bean *is actually created*. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created.

**Setter Dependency Injection (SDI) vs. Constructor Dependency Injection (CDI)**

|  |  |
| --- | --- |
| Setter DI | Constructor DI |
| Poor readability as it adds a lot of boiler plate codes in the application. | Good readability as it is separately present in the code. |
| The bean must include getter and setter methods for the properties. | The bean class must declare a matching constructor with arguments. Otherwise, BeanCreationException will be thrown. |
| Requires addition of @Autowired annotation, above the setter in the code and hence, it increases the coupling between the class and the DI container. | Best in the case of loose coupling with the DI container as it is not even required to add @Autowired annotation in the code) |
| Circular dependencies or partial dependencies result with Setter DI because object creation happens before the injections. | No scope for circular or partial dependency because dependencies are resolved before object creation itself. |
| Preferred option when properties are less and mutable objects can be created. | Preferred option when properties on the bean are more and immutable objects (eg: financial processes) are important for application. |