Spring and Spring Boot

# Dependency Injection & Loose Coupling

Suppose we want to implement binary search in our application. For that, let’s assume that we created a class as shown below.



The problem with above approach is that, each time you try to change the algorithm, you would need to change the code of binary search class. Is there any way to avoid it?

The solution to this is **interfaces**. Let’s write out two sorting algorithms separately, one is bubble sort and the other be quick sort.





Now, if we see carefully, the two algorithms override the sort method from sort algorithm interface. The interface is shown next.



Now, instead of hardcoding these sorting algorithms in the binary search, we can change the binary search class to as follows.



We see that we can initialize a private sort algorithm **dependency**, which we can initialize in the constructor. Inside the search method, we can simply use the sort algorithm dependency without anywhere referring to the actual sort algorithm. Now, the main method would look as follows.



If you see carefully, we only specify the algorithm when we initiate a binary search instance as a parameter to it.

## Spring Framework for automatic dependency injection using IoC

Till now we have done dependency injection using manual efforts. We created instances of binary search ourselves. However, in Spring, we can define **beans**, their dependencies and where to look for them.

We can define a bean using **@Component** annotation. We can define the dependencies using **@Autowired** annotation, and we do not need to define where to look out for these beans because **@SpringBootApplication** annotation automatically searches for beans in the directory and sub-directories where the main method is defined. Basically, @SpringBootApplication internally implements **@ComponentScan**.

We modify the main method as follows.



Let’s add binary search class and bubble sort as beans.





Inside binary search let us also add @Autowired to inject our sort algorithm interface. Basically, Spring will now automatically inject this dependency **using constructor**. Coming to the main method again,



Notice how we are storing the application context and then using **getBean** to fetch the singleton instance of binary search. However, also note that if we had added quick sort also as a bean, then the compiler would have given an error, asking for specifying a primary bean.

To solve this issue, make one bean as primary.

Now, we have made Bubble Sort as primary component and so adding @Component on multiple sorting algorithms won’t lead to any conflicts now.

## Setter Injection

In the previous section we saw constructor injection. There is one more way of injecting dependencies, and that is, setter injection.



Notice the name of the method where sort algorithm is getting bind. The setter has sort algorithm in its name. If we don’t use setter or constructor, by default setter injection is used (as shown below).



# Spring Modules



# Types of Spring Projects

1. **Spring Boot** – It is used for developing applications and microservices through auto-configuration and starter projects.
2. **Spring Cloud** – It is used for deploying and managing cloud native applications. We can deploy our spring boot applications on cloud through spring cloud.
3. **Spring Data** – It aims to solve consistency between various forms of databases.
4. **Spring Integration** – It aims to integrate enterprise integration patterns in your project.
5. **Spring Batch** – It is used to develop batch applications.
6. **Spring Security** – Provides solutions for securing your applications, eg. OAuth, etc.
7. **Spring HATEOAS**
8. **Others**

# Why Spring?

1. Enables writing testable code by providing support for dependency injection.
2. There is no **plumbing code**.
3. It has architecture flexibility. It can use MVC sometime and at other times it can use web services.
4. It can stay with trend. It is still relevant.