August 31, 2016 | Vojtech Ruzicka

Field Dependency Injection Considered Harmful

May 21, 2016 by

Field injection is a very popular practice in Dependency Injection frameworks, such as Spring. It has, however, several serious tradeoffs and should be generally avoided.

Injection Types

There are three main ways to inject your dependencies into your class. Constructor, Setter (Method) and Field injection. Let's quickly compare the code of the same dependencies injected by all the approaches.

Constructor

```
private DependencyA dependencyA;
private DependencyB dependencyB;
private DependencyC dependencyC;

@Autowired
public DI(DependencyA dependencyA, DependencyB dependencyB, DependencyC dependencyC) {
this.dependencyA = dependencyA;
this.dependencyB = dependencyB;
this.dependencyC = dependencyC;
}
```

Setter

```
private DependencyA dependencyA;
private DependencyB dependencyB;
private DependencyC dependencyC;
```

```
@Autowired
public void setDependencyA(DependencyA dependencyA) {
this.dependencyA = dependencyA;
}

@Autowired
public void setDependencyB(DependencyB dependencyB) {
this.dependencyB = dependencyB;
}

@Autowired
public void setDependencyC(DependencyC dependencyC) {
this.dependencyC = dependencyC;
}
```

Field

```
@Autowired
private DependencyA dependencyA;

@Autowired
private DependencyB dependencyB;

@Autowired
private DependencyC dependencyC;
```

What is wrong?

As you can see, the Field variant looks very nice. It is very short, concise, there is no boilerplate code. The code is easy to read and navigate. Your class can just focus on the important and is not polluted by DI boilerplate. You just put the @Autowired annotation above the fields and that's it. No special constructors or setters just for DI container to provide your dependencies. Java is very verbose as is, so every opportunity to make your code shorter is welcome, right?

Single Responsibility Principle Violation

It is very easy to add new dependencies. Maybe too easy. There is no problem in adding six, ten or even dozen dependencies. When you are using constructors for DI, after a certain point, the number of constructor params becomes too high and it is immediately obvious that something is wrong. Having too many dependencies usually means that the class has too many responsibilities. That may be violation of Single Responsibility Principle and separation of concerns and is a good indicator, that the class requires further inspection and possible refactoring. There is no such red flag when injecting directly to fields as this approach can scale indefinitely.

Dependency Hiding

Using DI container means that the class is no longer responsible for managing its own dependencies. Responsibility for obtaining the dependencies is extracted from the class. Someone other is now responsible for providing the dependencies – DI container or manually assigning them in tests. When the class is no longer responsible for obtaining its dependencies, it should clearly communicate them using public interface – methods or constructors. This way it is clear what the class requires and also whether it is optional (setters) or mandatory (constructors).

DI Container Coupling

One of the core ideas of the DI frameworks is that the managed class should have no dependency on the DI container used. In other words, it should be just a plain POJO, which can be instantiated independently, provided you pass it all the required dependencies. This way you can instantiate it in unit test without starting the DI container and test it separately (with container that would be more of integration test). If there is no container coupling, you can use the class either as managed or non-managed or even switch to a new DI framework.

However, when injecting directly to fields, you provide no direct way of instantiating the class with all its required dependencies. That means:

- There is a way (by calling the default constructor) to create object using *new* in a state when it is lacking some of its mandatory collaborators and usage will result in the NullPointerException.
- Such a class cannot be reused outside DI containers (tests, other modules) as there is no way except reflection to provide it with its required dependencies.

Immutability

Unlike constructor, field injection cannot be used to assign dependencies to final fields effectively rendering your objects mutable.

Constructor vs Setter Injection

So the Field injection may not be the way to go. What's left? Setters and Constructors. Which one should be used?

Setters

Setters should be used to inject optional dependencies. Class should be able to function when they are not provided. The dependencies can be changed anytime after the object is instantiated. That may on may not be an advantage depending on the circumstances. Sometimes it is desirable to have immutable object. Sometimes it is good to change the object's collaborators at runtime – such as JMX managed MBeans.

Official recommendation from Spring 3.x documentation encourages the use of setters over constructors:

> The Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy, especially when properties are optional. Setter methods also make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is a compelling use case.
>
> Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state. The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

Constructors

Constructor injection is good for mandatory dependencies. Those, which are required for the object to proper function. By supplying those in the constructor, you can be sure that the object is ready to be used the moment it is constructed. Fields assigned in the constructor can be also final, allowing object to be either completely immutable or at least protect its required fields.

One consequence of using constructor to provide dependencies is that circular dependency between two objects constructed in such way are no longer possible (unlike with setter injection). That is actually a good thing rather than limitation as circular dependencies should be avoided and are usually a sign of a bad design. This way such a practice is prevented.

Another advantage is that if using spring 4.3+, you can completely decouple your class from DI frameworks. The reason is that Spring now supports implicit constructor injection for one constructor scenarios. That means you no longer need DI annotations in your classes. Of course, you could achieve the same with explicitly configuring DI in your spring configs for given class, this just makes this whole lot easier.

As of Spring 4.x, the official recommendation from Spring documentation changes and setter injection is no longer encouraged over constructor:

> The Spring team generally advocates constructor injection as it enables one to implement application components as *immutable objects* and to ensure that required dependencies are not `null`. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a *bad code smell*, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.
>
> Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later.

Conclusion

Field injection should be mostly avoided. As a replacement, you should use either constructors or methods to inject your dependencies. Both have its advantages and disadvantages and the usage depends on the situation. However, as those approaches can be mixed, it is not either-or choice and you can combine both setter and constructor injection in one class. Constructors are more suitable for mandatory dependencies and when aiming for immutability. Setters are better for optional dependencies.

Filed Under: Spring
