**Spring field injection vs Constructor injection**

**Field injection**

This type of injection instruments some kind of reflection mechanism for injecting the required dependencies into the class.

While this injection type has the benefit, that it removes clutter code like setter methods or constructor parameters, it has the drawback that these dependencies are invisible. If you look at the class from the outside, you will only see the public methods and may be the constructor.

Even if this gives you a very clear idea what services a class provides, it has, in my opinion, this major drawback:

When writing tests for this particular class you have to inspect the class to see what are the required dependencies and must use either invoke the DI framework, even for simple tests, or use a kind of reflection mechanism to inject the dependencies (mocked / stubbed / real).

To make it even worse, the number of incoming dependencies is hidden within the class. Of course you can use tools (JDepend, etc. pp.) that tells you the number and direction of dependencies, or have a file that specifies the injection, but you must rely on this kind of tooling or inspect the class.

This is probably the most common form of **Dependency Injection**, because it reduces a lot of boilerplate code needed with setter or constructor injections.

**Spring bean to inject as dependency**

Let’s specify an interface for dependency:

package com.farenda.spring.tutorial.injection.field;  
  
  
public interface FoodRepository {  
  
String MenuById(int id);  
  
}

|  |
| --- |
|  |

And the actual Spring Bean implementation that will be injected:

package com.farenda.spring.tutorial.injection.field;  
  
  
import org.springframework.stereotype.Repository;  
  
  
import java.util.HashMap;  
  
import java.util.Map;  
  
  
@Repository  
  
public class InMemoryFoodRepository implements FoodRepository {  
  
  
// It's our local database ;-)  
  
private Map<Integer, String> Food = new HashMap<>();  
  
  
{  
  
Food.put(1, "Pizza");  
  
Food.put(2, "Pasta");  
  
Food.put(3, "Salad");  
  
}  
  
  
@Override  
  
public String MenuById(int id) {  
  
return Food.get(id);  
  
}  
  
}  
  
Spring component with dependency injected

|  |  |
| --- | --- |
| **using field injection**  Here we define another **Spring Bean**, but we are going to **inject** FoodRepository using **field injection**. To do that we have to annotate appropriate fields using **@Autowired** (or @Inject or @Resource) annotation:   |  | | --- | | package com.farenda.spring.tutorial.injection.field;   import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Component;   @Component  public class Library {   @Autowired  private FoodRepository FoodRepository;   public String findBook(int id) {  return FoodRepository.MenuById(id);  }  } |   It works without any setter, because underneath **Spring** is using **reflection** to inject dependencies.    **Constructor Injection**  Constructor injection instruments the constructor of a class, which used by the DI framework to inject the dependencies. It is the other variant that makes dependencies explicit.  *Each dependency is mandatory*  The benefits of this injection type are   1. You have to read exactly one method, the constructor, to figure out what are the dependencies of this class. 2. you create an immutable class which makes caching and so easier   The drawback here is, again, you can’t distinguish between optional and required dependencies. The constructor enforces that all fields are set.  Through the spring beans XML file you can configure your bean to initialize with an argument for the constructor, and then assign the arguments. Spring essentially “injects” the argument into your bean. This is referred to as **constructor injection**.  The following example passes in the String message using a constructor. The class is the same as the one in Basic Bean Creation except the default message on the message variable has been cleared and is now set to null. A single parameter constructor has been added to set a message.  **Example . ConstructorMessage**  public class ConstructorMessage {   private String message = null;   /\*\*  \* Constructor  \*/  public ConstructorMessage(String message) {  this.message = message;  }   /\*\*  \* Gets message.  \*/  public String getMessage() {  return message;  }   /\*\*  \* Sets message.  \*/  public void setMessage(String message) {  this.message = message;  }   }  The configuration for this bean is exactly the same as in the previous example, but now we have a new element, the *constructor-arg*. The *constructor-arg* element injects a message into the bean using the *constructor-arg* element's value attribute.  *ConstructorMessageTest-context.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"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd">   <bean id="message"  class="org.springbyexample.di.xml.ConstructorMessage">  <constructor-arg value="Spring is fun." />  </bean>   </beans>  **What to use / prefer?**  This answer depends heavily on your framework / team rule / language.  But I would strongly recommend to go for one of  the explicit injection variants, because they enable you to write pure unit tests without using the dependency injection framework. |