Spring Dependency Injection with Example

Last Updated : 30 Aug, 2025

Dependency Injection (DI) is a design pattern in which objects receive their dependencies from an external source rather than creating them internally. It promotes loose coupling, easier testing, and better code maintainability. In Spring, DI is achieved mainly through Constructor Injection or Setter Injection.

Need for Dependency Injection

- Suppose class One requires an object of class Two to perform its operations. It means class One is dependent on class Two.
- While such dependency may seem acceptable, in real-world applications, it can lead to Tight coupling, Difficulty in maintenance, reduced testability or potential system failures.
- Therefore, direct dependencies should be avoided.
- Spring IoC (Inversion of Control) resolves such dependency issues using Dependency Injection (DI).
- Dependency Injection makes the code easier to test and reuse.
- Loose coupling between classes is achieved by defining interfaces for common functionality or Letting the injector (Spring container) provide the appropriate implementation.
- The task of instantiating objects is done by the container according to the configurations specified by the developer.

Types of Spring Dependency Injection

There are two primary types of Spring Dependency Injection:

1. Setter Dependency Injection (SDI):

Setter DI involves injecting dependencies via setter methods. To configure SDI, the `@Autowired` annotation is used along with setter methods and the property is set through the `<property>` tag in the bean configuration file.

```
package com.geeksforgeeks.org;

import com.geeksforgeeks.org.IGeek;
import org.springframework.beans.factory.annotation.Autowired;

public class GFG {

    // The object of the interface IGeek
    private IGeek geek;

    // Setter method for property geek with @Autowired annotation
    @Autowired
    public void setGeek(IGeek geek) {
        this.geek = geek;
    }
}
```

Bean Configuration:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="GFG" class="com.geeksforgeeks.org.GFG">
        <property name="geek"  ref ="CsvGFG" />
    </bean>

<bean id="CsvGFG" class="com.geeksforgeeks.org.impl.CsvGFG" />
<bean id="JsonGFG" class="com.geeksforgeeks.org.impl.JsonGFG" />

</beans>
```

This injects the `CsvGFG` bean into the `GFG` object using the setter method (`setGeek`).

2. Constructor Dependency Injection (CDI):

Constructor DI involves injecting dependencies through constructors. To configure CDI, the `<constructor-arg>` tag is used in the bean configuration file.

```
package com.geeksforgeeks.org;

import com.geeksforgeeks.org.IGeek;

public class GFG {

    // The object of the interface IGeek
    private IGeek geek;

    // Constructor to set the CDI
    public GFG(IGeek geek) {
        this.geek = geek;
    }
}
```

Bean Configuration:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">


    <bean id="GFG" class="com.geeksforgeeks.org.GFG">
        <constructor-arg>
            <bean class="com.geeksforgeeks.org.impl.CsvGFG" />
        </constructor-arg>
    </bean>

<bean id="CsvGFG" class="com.geeksforgeeks.org.impl.CsvGFG" />
<bean id="JsonGFG" class="com.geeksforgeeks.org.impl.JsonGFG" />

</beans>
```

This injects the `CsvGFG` bean into the `GFG` object via the constructor.

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

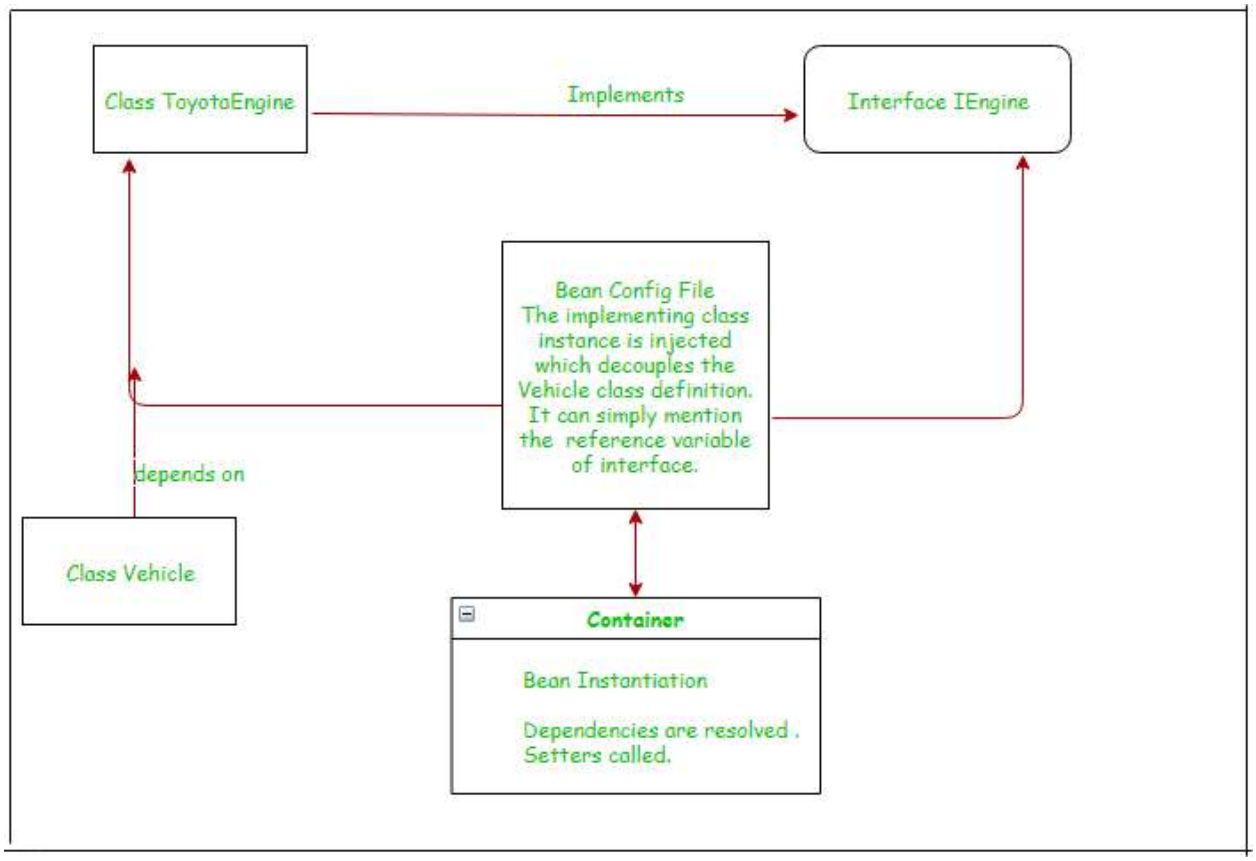
Setter DI	Constructor DI
Creates mutable objects. Dependencies can be modified after creation.	Creates immutable objects. Dependencies can't be modified after creation.
Dependencies can be injected later.	All dependencies must be provided at creation.
Requires addition of @Autowired annotation.	@Autowired annotation is not needed.
It results in circular dependencies or partial dependencies.	It too can have circular dependencies, it just fails faster and more explicitly.
Requires framework or manual setter calls for dependency injection in tests.	Easier unit testing - can create objects directly with mock dependencies.

Example of Spring DI

- We have 4 main components: **`IEngine`** interface, **`ToyotaEngine`** class (implements `IEngine`), **`Tyres`** class, **`Vehicle`** class (depends on `IEngine` and `Tyres`).
- Here, **Vehicle** is not creating it's own dependencies, Spring is handling object creating and wiring for us.
- Beans are defined in the XML configuration for: **ToyotaEngine,** Two **`Tyres`** instances (`tyre1Bean`, `tyre2Bean`), Two **`Vehicle`** instances.
- Two types of dependency injection are used: **InjectwithConstructor** uses constructor injection via **<constructor-arg>** tags and **InjectwithSetter** uses setter injection **via <property>** tags.
- Spring is determining which engine and tyres to inject into each **Vehicle**, based on the configuration.
- **Vehicle** is not aware of the actual implementation of the engine or tyre.

- This makes this system a loosely coupled system making it easier to maintain, more flexible, simpler to swap components without breaking the core logic.

Process Flow:



Code Implementation:

Enigne.java ToyotaEngine.java Tyres.java Vehicle.java pom.xml springContext.xml

```
interface IEngine {
    String EMISSION_NORMS = "BSIV";
    String importOrigin();
    double cost();
}
```