**Dependency Injection :** Dependency injection (DI) is a process whereby objects

define their dependencies, that is, the other objects they work with, only

through constructor arguments, arguments to a factory method, or properties

that are set on the object instance after it is constructed or returned from

a factory method. The container then injects those dependencies when it

creates the bean. This process is fundamentally the inverse, hence the name

Inversion of Control (IoC), of the bean itself controlling the instantiation

or location of its dependencies on its own by using direct construction of

classes, or the Service Locator pattern.

Code is cleaner with the DI principle and decoupling is more effective when

objects are provided with their dependencies. The object does not look up its

dependencies, and does not know the location or class of the dependencies. As

such, your classes become easier to test, in particular when the dependencies

are on interfaces or abstract base classes, which allow for stub or mock

implementations to be used in unit tests.

DI exists in two major variants, Constructor-based dependency injection and

Setter-based dependency injection.

**Constructor-based dependency injection :** Constructor-based DI is accomplished

by the container invoking a constructor with a number of arguments, each

representing a dependency. Calling a static factory method with specific

arguments to construct the bean is nearly equivalent, and this discussion

treats arguments to a constructor and to a static factory method similarly.

The following example shows a class that can only be dependency-injected with

constructor injection. Notice that there is nothing special about this class,

it is a POJO that has no dependencies on container specific interfaces, base

classes or annotations.

**CODE SNIPPET :** public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on a MovieFinder private

MovieFinder movieFinder;

// a constructor so that the Spring container can inject a MovieFinder public

SimpleMovieLister(MovieFinder movieFinder) { this.movieFinder = movieFinder;

}

// business logic that actually uses the injected MovieFinder is omitted... }

**Constructor argument resolution :** 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. Consider the following class

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) { // ... } }

No potential ambiguity exists, assuming that Bar and Baz classes are not

related by inheritance. Thus the following configuration works fine, and you

do not need to specify the constructor argument indexes and/or types

explicitly in the <constructor-arg/> element.

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

When another bean is referenced, the type is known, and matching can occur

(as was the case with the preceding example). When a simple type is used,

such as <value>true</value>, Spring cannot determine the type of the value,

and so cannot match by type without help. Consider the following class:

package examples;

public class ExampleBean {

// Number of years to calculate the Ultimate Answer private int years;

// The Answer to Life, the Universe, and Everything private String

ultimateAnswer;

public ExampleBean(int years, String ultimateAnswer) { this.years = years;

this.ultimateAnswer = ultimateAnswer; } }

<bean id="exampleBean" class="examples.ExampleBean"> <constructor-arg

type="int" value="7500000"/> <constructor-arg type="java.lang.String"

value="42"/> </bean>

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>

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>

**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.

**CODE SNIPPET**

public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on the MovieFinder private

MovieFinder movieFinder;

// a setter method so that the Spring container can inject a MovieFinder

public void setMovieFinder(MovieFinder movieFinder) { this.movieFinder =

movieFinder; }

// business logic that actually uses the injected MovieFinder is omitted...

}

The ApplicationContext supports constructor-based and setter-based DI for the beans it manages.

It also supports setter-based DI after some dependencies have already been injected through the constructor approach.

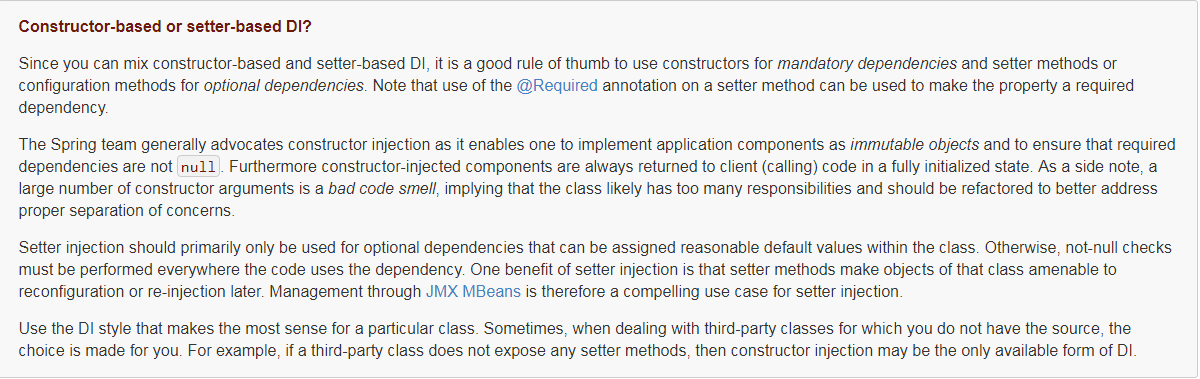
You configure the dependencies in the form of a BeanDefinition, which you use in conjunction with PropertyEditor

instances to convert properties from one format to another. However, most Spring users do not work with these classes

directly (i.e., programmatically) but rather with XML bean definitions, annotated components (i.e., classes annotated with @Component, @Controller, etc.),

or @Bean methods in Java-based @Configuration classes. These sources are then converted internally into instances of BeanDefinition and used to load an

entire Spring IoC container instance.



**Dependency resolution process**

The container performs bean dependency resolution as follows:

* 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.
* 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.
* Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
* 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.

**Examples of dependency injection**

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions:

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

*<!-- setter injection using the nested ref element -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter injection using the neater ref attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

**this**.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

**this**.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

**this**.i = i;

}

}

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

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

*<!-- constructor injection using the nested ref element -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- constructor injection using the neater ref attribute -->*

<constructor-arg ref="yetAnotherBean"/>

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

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

**this**.beanOne = anotherBean;

**this**.beanTwo = yetAnotherBean;

**this**.i = i;

}

}

Now consider a variant of this example, where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object:

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

*// a private constructor*

**private** ExampleBean(...) {

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public** **static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// some other operations...*

**return** eb;

}

}

Arguments to the static factory method are supplied via <constructor-arg/> elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the static factory method, although in this example it is.