Spring IoC Container

### Introduction to the Spring IoC container and beans

Inversion of Control (IoC) [[1](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "_footnote_1)] principle is also known as dependency injection (DI). It 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 by using direct construction of classes.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container. The [BeanFactory](https://docs.spring.io/spring-framework/docs/5.0.6.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.6.RELEASE/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds easier integration with Spring’s AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications. In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

### Container overview

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the web.xml file of the application will typically suffice.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.

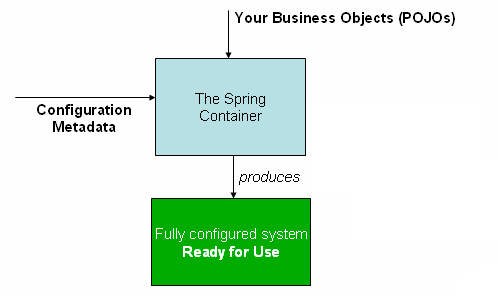


Figure 1. The Spring IoC container

#### Configuration metadata

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as <bean/> elements inside a top-level <beans/> element. Java configuration typically uses @Bean annotated methods within a @Configuration class.

The following example shows the basic structure of XML-based configuration metadata:

**<?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="..." class="...">**

**<!-- collaborators and configuration for this bean go here -->**

**</bean>**

**<bean id="..." class="...">**

**<!-- collaborators and configuration for this bean go here -->**

**</bean>**

**<!-- more bean definitions go here -->**

**</beans>**

The id attribute is a string that you use to identify the individual bean definition. The class attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects.

#### Instantiating a container

The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

**ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");**

The following example shows the service layer objects (services.xml) configuration file:

**<?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">**

**<!-- services -->**

**<bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">**

**<property name="accountDao" ref="accountDao"/>**

**<property name="itemDao" ref="itemDao"/>**

**<!-- additional collaborators and configuration for this bean go here -->**

**</bean>**

**<!-- more bean definitions for services go here -->**

**</beans>**

The following example shows the data access objects daos.xml file:

**<?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="accountDao"**

**class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">**

**<!-- additional collaborators and configuration for this bean go here -->**

**</bean>**

**<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">**

**<!-- additional collaborators and configuration for this bean go here -->**

**</bean>**

**<!-- more bean definitions for data access objects go here -->**

**</beans>**

In the preceding example, the service layer consists of the class PetStoreServiceImpl, and two data access objects of the type JpaAccountDao and JpaItemDao (based on the JPA Object/Relational mapping standard). The property name element refers to the name of the JavaBean property, and the ref element refers to the name of another bean definition. This linkage between id and ref elements expresses the dependency between collaborating objects.

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture. You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files. For example:

**<beans>**

**<import resource="services.xml"/>**

**<import resource="resources/messageSource.xml"/>**

**<import resource="/resources/themeSource.xml"/>**

**<bean id="bean1" class="..."/>**

**<bean id="bean2" class="..."/>**

**</beans>**

The import directive is a feature provided by the beans namespace itself. The contents of the files being imported, including the top level <beans/> element, must be valid XML bean definitions according to the Spring Schema.

#### Using the container

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method T getBean(String name, Class<T> requiredType) you can retrieve instances of your beans.

The ApplicationContext enables you to read bean definitions and access them as follows:

**// create and configure beans**

**ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");**

**// retrieve configured instance**

**PetStoreService service = context.getBean("petStore", PetStoreService.class);**

**// use configured instance**

**List<String> userList = service.getUsernameList();**

### Bean overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions. Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain the following metadata:

* A package-qualified class name: typically the actual implementation class of the bean being defined.
* Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
* References to other beans that are needed for the bean to do its work; these references are also called collaborators or dependencies.
* Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the ApplicationContext’s BeanFactory via the method getBeanFactory() which returns the BeanFactory implementation DefaultListableBeanFactory. DefaultListableBeanFactory supports this registration through the methods registerSingleton(..) and registerBeanDefinition(..).

#### Naming beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). The id attribute allows you to specify exactly one id. Conventionally these names are alphanumeric ('myBean', 'fooService', etc.), but may contain special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the name attribute, separated by a comma (,), semicolon (;), or white space.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the ref element or [Service Locator](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-servicelocator) style lookup, you must provide a name.

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) 'accountService', 'userDao', and so forth.

With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules above: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved.

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the <alias/> element to accomplish this.

**<alias name="fromName" alias="toName"/>**

In this case, a bean in the same container which is named fromName, may also, after the use of this alias definition, be referred to as toName.

For example, the configuration metadata for subsystem A may refer to a DataSource via the name subsystemA-dataSource. The configuration metadata for subsystem B may refer to a DataSource via the name subsystemB-dataSource. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name myApp-dataSource. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

**<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>**

**<alias name="subsystemA-dataSource" alias="myApp-dataSource" />**

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

#### Instantiating beans

A bean definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the class attribute of the <bean/> element. This class attribute, which internally is a Class property on a BeanDefinition instance, is usually mandatory. You use the Class property in one of two ways:

* Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the new operator.
* To specify the actual class containing the static factory method that will be invoked to create the object, in the less common case where the container invokes a static factory method on a class to create the bean. The object type returned from the invocation of the static factory method may be the same class or another class entirely.

If you want to configure a bean definition for a static nested class, you have to use the binary name of the nested class.

For example, if you have a class called Foo in the com.example package, and this Foo class has a static nested class called Bar, the value of the 'class' attribute on a bean definition would be...​

**com.example.Foo$Bar**

Notice the use of the $ character in the name to separate the nested class name from the outer class name.

##### Instantiation with a constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually any class you want it to manage; it is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container.

With XML-based configuration metadata you can specify your bean class as follows:

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

**<bean name="anotherExample" class="examples.ExampleBeanTwo"/>**

##### Instantiation with a static factory method

When defining a bean that you create with a static factory method, you use the class attribute to specify the class containing the static factory method and an attribute named factory-method to specify the name of the factory method itself. You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the createInstance() method must be a static method.

**<bean id="clientService"**

**class="examples.ClientService"**

**factory-method="createInstance"/>**

**public class ClientService {**

**private static ClientService clientService = new ClientService();**

**private ClientService() {}**

**public static ClientService createInstance() {**

**return clientService;**

**}**

**}**

##### Instantiation using an instance factory method

Similar to instantiation through a [static factory method](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-factory-class-static-factory-method), instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the class attribute empty, and in the factory-bean attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute.

**<!-- the factory bean, which contains a method called createInstance() -->**

**<bean id="serviceLocator" class="examples.DefaultServiceLocator">**

**<!-- inject any dependencies required by this locator bean -->**

**</bean>**

**<!-- the bean to be created via the factory bean -->**

**<bean id="clientService"**

**factory-bean="serviceLocator"**

**factory-method="createClientServiceInstance"/>**

**public class DefaultServiceLocator {**

**private static ClientService clientService = new ClientServiceImpl();**

**public ClientService createClientServiceInstance() {**

**return clientService;**

**}**

**}**

One factory class can also hold more than one factory method as shown here:

**<bean id="serviceLocator" class="examples.DefaultServiceLocator">**

**<!-- inject any dependencies required by this locator bean -->**

**</bean>**

**<bean id="clientService"**

**factory-bean="serviceLocator"**

**factory-method="createClientServiceInstance"/>**

**<bean id="accountService"**

**factory-bean="serviceLocator"**

**factory-method="createAccountServiceInstance"/>**

**public class DefaultServiceLocator {**

**private static ClientService clientService = new ClientServiceImpl();**

**private static AccountService accountService = new AccountServiceImpl();**

**public ClientService createClientServiceInstance() {**

**return clientService;**

**}**

**public AccountService createAccountServiceInstance() {**

**return accountService;**

**}**

**}**

### Dependencies

#### 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.

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.

DI exists in two major variants, [Constructor-based dependency injection](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-constructor-injection) and [Setter-based dependency injection](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-setter-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. 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.

**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.

**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.

**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;**

**}**

**}**

##### Constructor argument type matching

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute.

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

**<constructor-arg type="int" value="7500000"/>**

**<constructor-arg type="java.lang.String" value="42"/>**

**</bean>**

##### Constructor argument index

Use the index attribute to specify explicitly the index of constructor arguments.

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

**<constructor-arg index="0" value="7500000"/>**

**<constructor-arg index="1" value="42"/>**

**</bean>**

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the index is 0 based.

##### Constructor argument name

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>**

Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can’t compile your code with debug flag (or don’t want to) you can use [@ConstructorProperties](http://download.oracle.com/javase/6/docs/api/java/beans/ConstructorProperties.html) JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

**package examples;**

**public class ExampleBean {**

**// Fields omitted**

**@ConstructorProperties({"years", "ultimateAnswer"})**

**public ExampleBean(int years, String ultimateAnswer) {**

**this.years = years;**

**this.ultimateAnswer = ultimateAnswer;**

**}**

**}**

##### 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.

The following example shows a class that can only be dependency-injected using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

**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.

**Constructor-based or setter-based DI?**

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the [@Required](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-required-annotation) annotation on a setter method can be used to make the property a required dependency.

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. Management through [JMX MBeans](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/integration.html" \l "jmx) is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

##### 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.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Bean scopes](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-factory-scopes). Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late, i.e. on first creation of the affected bean.

**Circular dependencies?**

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. For example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why ApplicationContext implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the ApplicationContext is created, not later. You can still override this default behavior so that singleton beans will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a [configured init method](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-factory-lifecycle-initializingbean) or the [InitializingBean callback method](https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/core.html" \l "beans-factory-lifecycle-initializingbean)) are invoked.

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;**

**}**

**}**

The constructor arguments specified in the bean definition will be used as arguments to the constructor of the ExampleBean.

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. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the factory-bean attribute instead of the class attribute).

#### Dependencies and configuration in detail